Programmation par contraintes sur les flux de données

Xavier Dupont

Université de Caen Basse-Normandie

École doctorale SIMEM

Thèse de doctorat

présentée et soutenue le : *18/12/2014*

par

Xavier Dupont

pour obtenir le

Doctorat de l'Université de Caen Basse-Normandie

Spécialité : Informatique et applications

Programmation par Contraintes sur les Flux de Données

Directeur de thèse : *Arnaud Lallouet*

Jury

Éric Monfroy	Professeur	Université de Nantes	(Rapporteur)
Frédéric Saubion	Professeur	Université d'Angers	(Rapporteur)
Patrice Boizumault	Professeur	Université de Caen	(Examinateur)
Arnaud Lallouet	Professeur	Université de Caen	(Directeur)

Table des matières

Remerciements

Je remercie tout d'abord mon directeur de thèse Mr. Arnaud Lallouet pour m'avoir permis de travailler en autonomie sur un sujet novateur, en me laissant une grande liberté de manœuvre concernant le déroulement et l'organisation de mes recherches.

J'adresse également mes remerciements aux rapporteurs Mr. Eric Monfroy et Mr. Frédéric Saubion, pour avoir accepté d'étudier mon travail, et Mr. Patrice Boizumault pour avoir été présent depuis le début, et pour assister en tant qu'examinateur au résultat de mes efforts.

Un tel travail est nécessairement ponctué de périodes de doutes intenses, et n'aurait peut-être pas été mené à terme sans le soutien de nombreux amis et collègues. Je tiens en premier lieu à remercier mes collègues de thèses et amis Mehdi Khiary, Mariya Georgieva et Mathieu Roux, qui ont toujours été à mes côtés, avec qui nous avons pratiquement été inséparables, et qui ont chacuns été présents lors de périodes difficiles. Je tiens aussi à remercier François Rioult pour avoir cru en moi depuis la licence d'informatique jusqu'à la fin de la thèse.

Cette thèse a duré quatre ans, dont une année en ATER plein, qui a été plus particulière que les autres, et je remercie le personnel de l'Ensicaen pour m'avoir permis d'être opérationnel en moins d'une semaine malgré le nombre de formalités à effectuer et la complexité de la gestion des emploi du temps.

Je remercie le personnel du GREYC pour m'avoir fourni un environnement de travail et des collègues agréables, et je remercie grandement le personnel de l'école doctorale pour m'avoir suivi pendant ces quatres années.

Enfin, je remercie ma famille pour m'avoir toujours soutenu.

Chapitre 1

Introduction

La programmation par contraintes est un paradigme déclaratif pour la résolution de problèmes combinatoires. Son application aux problèmes de planification est fréquente, mais oblige l'utilisateur à utiliser une borne artificielle sur l'horizon temporel considéré. De façon plus générale, les techniques de programmation par contraintes actuelles ne permettent pas de traiter aisément des problèmes portant sur un horizon temporel infini.

Nous proposons le cadre des StCSPs pour résoudre des problèmes de programmation par contraintes sur les variables flux. Ce type de variable vient s'ajouter aux types de variables déjà traités par la programmation par contraintes, tels que les variables à domaines finis, les flottants, et les graphes, entre autres. Nous présentons une famille de contraintes temporelles qui permet d'utiliser directement les techniques de propagation existantes en programmation par contraintes pour produire des solutions de problèmes de satisfaction sur les variables flux. Contrairement à la plupart des formalismes à bases de contraintes qui considèrent des séquences infinies, notre formalisme permet de générer des solutions arbitraires, alors que d'autres se restreignent à des solutions de formes particulières, telles que des solutions ultimement périodiques.

Une part importante du travail à été dévolue à la comparaison de notre méthode avec les méthodes de *model checking*. Ces méthodes sont basées sur des logiques temporelles, qui permettent d'exprimer des spécifications logiques temporelles, généralement propositionnelles. Les logiques temporelles sont très nombreuses et diversifiées, et les méthodes algorithmiques associées sont toujours en évolution. Cependant, nous montrerons que cette diversité cache des caractéristiques communes, et qu'il est possible de passer d'une logique temporelle à une autre par l'ajout ou la suppression de certaines caractéristiques. Nous motiverons ainsi le choix de la logique temporelle linéaire propositionnelle comme logique temporelle de référence, et seront en mesure de donner des pistes pour la généralisation des présents résultats à d'autres logiques.

Notre travail se place à la frontière entre programmation par contraintes et *model checking*. Les liens entre ces deux disciplines sont relativement peu explorés. Bien que certains travaux utilisent les deux [49, 37], l'utilisation de la programmation par contraintes en *model checking*, ou inversement, reste assez marginale et souvent limitée à des contextes applicatifs très spécifiques. Le présent travail vise une étude plus systèmatique de l'applicabilité de la programmation par contraintes dans un cadre temporel au sens des logiques temporelles.

Première partie

État de l'art

Chapitre 2

Notations et définitions préliminaires

Sommaire

Nous définissons dans ce chapitre les différentes notions et notations qui seront utilisées dans la suite.

$\mathbb{N}$ dénote l'ensemble des entiers naturels. Pour $a \in \mathbb{N}$ et $b \in \mathbb{N}$, $\{a, \ldots, b\}$ dénote l'ensemble $\{k \in \mathbb{N} \mid a \leqslant k \wedge k \leqslant b\}$.

$\mathbb{B} = \{\bot, \top\}$ dénote l'ensemble des booléens. Pour une propriété P, la notation $[P]$ représente la valeur de vérité de P. En plus des quantification existentielle ($\exists$) et universelle ($\forall$), nous utilisons le symbole '!' pour dénoter un élément qui peut être choisi d'une seule façon. Par exemple, $!n \in \mathbb{N}, \forall n' \in \mathbb{N}, n \leqslant n'$ dénote l'entier naturel 0.

2.1 Ensembles, tuples, séquences

Nous distinguons les notions d'ensemble, de tuple et de séquence.

2.1.1 Ensembles

Un **ensemble** E est une collection non-ordonnée d'objets, appelés **éléments** de E.

Soient A, B, C des ensembles, et a, b, c des éléments. Les notations pour les ensembles sont les suivantes :

— "$a \in A$" signifie que a est un élément de A ;
— "$A \subset B$" signifie que A est inclus dans, mais différent de, B ;
— "$A \subseteq B$" signifie que A est inclus dans B ;
— "$|A|$" dénote la cardinalité de A
Si A contient un nombre fini d'éléments, sa cardinalité est le nombre de ses éléments
Si A contient une quantité dénombrable d'éléments, $|A| = \aleph_0$
Si A contient une quantité indénombrable d'éléments, $|A| = \aleph_1$
— "$A \cup B$" dénote l'union de A et B ;
— "$A \cap B$" dénote l'intersection de A et B ;
— "$A \backslash B$" dénote la différence de A et B ;
— "Part(A)" représente l'ensemble des sous-ensembles de A.

Dans les algorithmes, nous utiliserons également les procédures suivantes :
— "`AJOUTER`$(!A, B)$" ajoute à l'ensemble A tous les éléments de l'ensemble B ;
— "`SUPPRIMER`$(!A, B)$" enlève de l'ensemble A tous les éléments présents dans l'ensemble B.

2.1.2 Tuple

Un **tuple** τ sur des ensembles E_1, ..., E_k est un élément $\langle \tau_1, \ldots, \tau_k \rangle$ de $E_1 \times \cdots \times E_k$.

Les notations sur les tuples sont les suivantes :
— La notation "$|\tau|$" dénote la taille k de τ ;
— La notation "$\tau[i]$" dénote la i-ème composante de τ, c'est-à-dire que $\tau[i] \in E_i$.

Les tuples servent principalement à décrire des objets structurés, pour lesquels il est préférable de définir des fonctions spécifiques pour identifier les différentes composantes.

2.1.3 Séquence

Une **séquence** σ d'index I sur un ensemble E est une fonction totale $I \to E$ avec I un segment de $\mathbb{N}$ tel que $I = [1 \ldots k]$ avec $k \in \mathbb{N}$ ou $I = \mathbb{N}$. La notation $\sigma = \langle \sigma[1], \sigma[2], \ldots, \rangle$ est communément employée pour décrire une séquence.

Les notations sur les séquences sont les suivantes :
— "$|\sigma|$" dénote la longueur de σ, c'est-à-dire la cardinalité de l'index de σ ;
— "$\sigma[i]$" dénote le i-ème élément de σ ;
— "$\sigma \cdot \sigma'$" dénote la concaténation de σ avec σ', ce qui est possible seulement si $|\sigma| < \aleph_0$

$$(\sigma \cdot \sigma')[i] = \begin{cases} \sigma[i] & 1 \leqslant i \leqslant |\sigma| \\ \sigma'[i - |\sigma|] & |\sigma| < i \leqslant |\sigma'| \end{cases}$$

— "Inf(σ)" représente l'ensembles des symboles de Σ qui apparaissent une infinités de fois dans σ

$$\mathrm{Inf}(\sigma) = \{s \in \Sigma \mid \forall n > 0, \exists k \in \{n+1, \ldots, |\sigma|\}, \sigma[k] = s\}$$

Le terme de **mot** sur un **alphabet** Σ est un synonyme fréquemment employé en théorie des langages.

Soit Σ un ensemble. Les notations suivantes sont fréquemment utilisées pour désigner des ensembles de séquences particuliers :

— Σ^k représente l'ensemble des séquences de longueur k sur Σ ;

— Σ^* représente l'ensemble des séquences finies sur Σ ;

— Σ^ω représente l'ensemble des séquences infinies sur Σ ;

— $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ représente l'ensemble de toutes les séquences sur Σ

2.2 Graphes orientés

Un graphe orienté est un ensemble de nœuds qui sont reliés par des arêtes.

Définition 1. Un **graphe orienté** est tuple $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$ dans lequel :

— $\mathcal{N}$ est l'ensemble des nœuds de $\mathcal{G}$

— $\mathcal{E} \in \mathcal{N}^2$ est l'ensemble des arêtes

Un nœud $n \in \mathcal{N}$ est **parent** d'un nœud n' si $\langle n, n' \rangle \in \mathcal{E}$. Un nœud $n \in \mathcal{N}$ est **enfant** d'un nœud n' si $\langle n', n \rangle \in \mathcal{E}$. Un graphe est **non-orienté** si tout nœud n' enfant d'un nœud n est aussi parent de n. Un **nœud racine**, ou **nœud source**, est un nœud qui n'a pas de parents. Un **nœud feuille**, ou **nœud puits**, est un nœud qui n'a pas d'enfants. Un **nœud interne** est un nœud qui n'est ni un nœud racine, ni un nœud feuille. Le **degré** d'un nœud est le nombre d'enfants de ce nœud. Le **degré d'un graphe** est le nombre maximum d'enfants que peut avoir un nœud dans ce graphe. Un **chemin orienté** d'un nœud n à un nœud n' est une séquence $\nu \in \mathcal{N}^*$ telle que $\nu[1] = n$, $\nu[|\nu|] = n'$, et pour chaque $i \in [2 \dots |\nu|]$, $\nu[i-1, i] \in \mathcal{E}$. Un nœud n' est **descendant** d'un nœud n s'il existe un chemin orienté de n à n'. Un nœud n' est **ancêtre** d'un nœud n s'il existe un chemin orienté de n' à n.

Comme les graphes et chemins non-orientés sont des cas particuliers des graphes et chemins orientés, nous considérerons dans la suite qu'un graphe ou un chemin est supposé orienté, sauf mention explicite contraire.

Soit $\mathcal{G}$ un graphe et n un nœud de $\mathcal{G}$. Nous utiliserons dans les algorithmes les prédicats et opérations suivants sur les graphes :

— NOEUDS($\mathcal{G}$) représente l'ensemble des nœuds $\mathcal{N}$ de $\mathcal{G}$

— ARETES($\mathcal{G}$) représente l'ensemble des arêtes $\mathcal{E}$ de $\mathcal{G}$

— PARENTS($n, \mathcal{G}$) représente l'ensemble des nœuds parents de n

$$\text{PARENTS}(n, \mathcal{G}) = \{n' \in \text{NOEUDS}(\mathcal{G}) \mid \langle n', n \rangle \in \text{ARETES}(\mathcal{G})\}$$

— ENFANTS($n, \mathcal{G}$) représente l'ensemble des nœuds enfants de n

$$\text{ENFANTS}(n, \mathcal{G}) = \{n' \in \text{NOEUDS}(\mathcal{G}) \mid \langle n, n' \rangle \in \text{ARETES}(\mathcal{G})\}$$

— RACINES($n, \mathcal{G}$) représente l'ensemble des nœuds racine de $\mathcal{G}$

$$\text{RACINES}(\mathcal{G}) = \{n \in \text{NOEUDS}(\mathcal{G}) \mid \text{PARENTS}(n, \mathcal{G}) = \emptyset\}$$

— FEUILLES($\mathcal{G}$) représente l'ensemble des nœuds feuille de $\mathcal{G}$

$$\text{FEUILLES}(\mathcal{G}) = \{n \in \text{NOEUDS}(\mathcal{G}) \mid \text{ENFANTS}(n, \mathcal{G}) = \emptyset\}$$

— INTERNES($\mathcal{G}$) représente l'ensemble des nœuds internes de $\mathcal{G}$

$$\text{INTERNES}(\mathcal{G}) = \text{NOEUDS}(\mathcal{G}) \backslash (\text{RACINES}(\mathcal{G}) \cup \text{FEUILLES}(\mathcal{G}))$$

— CHEMINS($n, \mathcal{G}$) représente l'ensemble des chemins dans $\mathcal{G}$ partant de n

$$\begin{aligned}\text{CHEMINS}(n, \mathcal{G}) = \{&\nu \in \text{NOEUDS}(\mathcal{G})^\infty \mid \nu[1] = n \\ &\wedge \forall i \in [2 \dots [\nu]], \langle \nu[i-1], \nu[i] \rangle \in \text{ARETES}(\mathcal{G})\}\end{aligned}$$

— CHEMINS($n, n', \mathcal{G}$) représente l'ensemble des chemins de n à n' dans $\mathcal{G}$

$$\text{CHEMINS}(n, n', \mathcal{G}) = \{\nu \in \text{CHEMINS}(\mathcal{G}) \cap \text{NOEUDS}(\mathcal{G})^* \mid \nu[1] = n \wedge \nu[|\nu|] = n'\}$$

— DESCENDANTS($n, \mathcal{G}$) représente l'ensemble des descendants de n dans $\mathcal{G}$

$$\begin{aligned}\text{DESCENDANTS}(n, \mathcal{G}) = \{&n' \in \text{NOEUDS}(\mathcal{G}) \mid \\ &\exists \nu \in \text{CHEMINS}(n, \mathcal{G}), \exists i \in [2, |\nu|], \nu[i] = n'\}\end{aligned}$$

— ANCETRES($n, \mathcal{G}$) représente l'ensemble des ancêtres de n dans $\mathcal{G}$

$$\text{ANCETRES}(n, \mathcal{G}) = \{n' \in \text{NOEUDS}(\mathcal{G}) \mid \exists \nu \in \text{CHEMINS}(n', \mathcal{G}), \exists i \in [2, |\nu|], \nu[i] = n\}$$

Etant donné un ensemble de graphes $\mathcal{G}$, nous définissons $\text{NOEUDS}_\forall(\mathcal{G})$ l'ensembles des nœuds de tous les graphes :

$$\text{NOEUDS}_\forall(\mathcal{G}) = \{\langle G, n \rangle \mid G \in \mathcal{G} \wedge n \in \text{NOEUDS}(G)\}$$

et étant donné $\langle G, n \rangle \in \text{NOEUDS}(\mathcal{G})$, nous posons :

— $\text{GRAPHE}(\langle G, n \rangle) = G$

— $\text{NOEUD}(\langle G, n \rangle) = n$

Définition 2 (Graphe étiqueté)**.** Un **graphe étiqueté** est un tuple $\mathcal{G}_\mathcal{E} = \langle \mathcal{G}, \ell_\mathcal{N}, \ell_\mathcal{E}, E_\mathcal{N}, E_\mathcal{E} \rangle$ dans lequel :

— $\mathcal{G}$ est le **graphe support** de $\mathcal{G}_\mathcal{E}$;

— $E_\mathcal{N}$ est l'ensemble des étiquettes des nœeuds ;

— $E_\mathcal{E}$ est l'ensemble des étiquettes des arêtes ;

— $\ell_\mathcal{N} : \text{NOEUDS}(\mathcal{G}) \to E_\mathcal{N}$ est la fonction d'étiquetage des nœuds ;

— $\ell_\mathcal{E} : \text{ARETES}(\mathcal{G}) \to E_\mathcal{E}$ est la fonction d'étiquetage des nœuds.

Les opérations sur les graphes, lorsqu'elles sont appliquées à un graphe étiqueté, s'appliquent à son graphe support. Pour un graphe dont seulement les nœuds ou bien les arêtes sont étiquetés, les éléments non utilisés sont laissés non spécifiés.

Nous procédons maintenant aux définitions de quelques graphes aux propriétés particulières.

Définition 3 (Graphe dirigé acyclique)**.** Un **graphe dirigé acyclique** est un graphe qui ne contient pas de cycles, c'est-à-dire un graphe $\mathcal{G}$ dans lequel il existe au plus un chemin entre chaque nœud :

$$\forall n, n' \in \text{NOEUDS}(\mathcal{G}), |\text{CHEMINS}(n, n', \mathcal{G})| \leqslant 1$$

Définition 4 (Forêt)**.** Une **forêt** est un graphe dirigé acyclique $\mathcal{F}$ dans lequel chaque nœud a au plus un parent :

$$\forall n \in \text{NOEUDS}(\mathcal{F}), |\text{PARENTS}(n, \mathcal{F})| \leqslant 1$$

Pour une forêt $\mathcal{F}$, nous définissnos la fonction partielle $\text{PARENT}(n, \mathcal{F})$ qui renvoit l'unique parent d'un nœud n dans F, si n n'est pas un nœud racine.

Définition 5 (Arbre)**.** Un **arbre** est une forêt $\mathcal{T}$ qui possède une unique racine :

$$|\text{RACINES}(\mathcal{T})| = 1$$

L'unique racine d'un arbre est notée $\text{RACINE}(\mathcal{T})$.

2.2.1 Composantes fortement connexes

La notion de composante fortement connexe permet de résumer la structure d'un graphe orienté.

Définition 6 (Composante fortement connexe)**.** Soit $\mathcal{G}$ un graphe. Un sous-ensemble $N \subseteq \text{NOEUDS}(\mathcal{G})$ est **fortement connexe** s'il existe un chemin entre toute paire de nœuds de N, c'est-à-dire si :

$$\text{FC}(N) \equiv \forall n, n' \in N, |\text{CHEMINS}(n, n', \mathcal{G})| \geqslant 0$$

Un sous-ensemble fortement connexe $N \subseteq \text{NOEUDS}$ est une **composante fortement connexe** s'il est maximal, c'est-à-dire si la condition suivante est satisfaite :

$$\forall n' \in \text{NOEUDS}(\mathcal{G}) \backslash N, \forall n \in N, |\text{CHEMINS}(n, n', \mathcal{G})| = 0 \vee |\text{CHEMINS}(n', n, \mathcal{G})| = 0$$

Le cas particuliers des singletons mérite l'attention. Pour un graphe $\mathcal{G}$, nous considérons qu'un singleton $\{n\}$ est une composante fortement connexe si et seulement si la définition précédente est satisfaite, ce qui impose que $\langle n, n \rangle \in \text{ARETES}(\mathcal{G})$.

Un résultat classique est qu'à tout graphe orienté sans boucles correspond le graphe dirigé acyclique de ses composantes fortement connexes [132], ce qui revient en réalité à considérer qu'une boucle est implicitement présente sur chaque nœud. Nous introduisons la définition de "composante fortement connexe singulière" pour ne pas enlever à la beauté du résultat, tout en prenant en compte cette particularité des singletons.

Définition 7 (Composante fortement connexe singulière)**.** Soit $\mathcal{G}$ un graphe, et $n \in \text{NOEUDS}(\mathcal{G})$ un nœud de $\mathcal{G}$ tel que pour toute composante fortement connexe $C \subseteq \text{NOEUDS}(\mathcal{G})$ au sens de la définition 6, $n \notin C$. Alors nous considérons que $\{n\}$ est une **composante fortement connexe singulière**.

Nous notons $\text{CFC}(\mathcal{G})$ l'ensemble des composantes fortement connexes d'un graphe $\mathcal{G}$, et nous introduisons le prédicat $\text{SINGULIERE}(C, \mathcal{G})$ qui est vrai si et seulement si C est une composante fortement connexe singulière.

Le graphe des composantes fortement connexes est défini de la façon suivante :

Définition 8 (Graphe des composantes fortement connexes)**.** Soit $\mathcal{G}$ un graphe. Le graphe des composantes fortement connexes de $\mathcal{G}$ est le graphe $\mathcal{G}_{\text{CFC}}$ dans lequel :

— $\text{NOEUDS}(\mathcal{G}_{\text{CFC}}) = \text{CFC}(\mathcal{G})$;

— $\langle C_1, C_2\rangle \in \text{ARETES}(\mathcal{G}_{\text{CFC}})$ si et seulement si :
$\exists n_1 \in C_1, \exists n_2 \in C_2, \langle n_1, n_2\rangle \in \text{ARETES}(\mathcal{G})$.

A partir du graphe des composantes fortement connexes, nous pouvons définir de façon naturelle les notions de composante fortement connexes racine, internes, et feuille. Noter que toutes les composantes fortement connexes du graphe des composante fortement connexes sont nécessairement singulières.

2.2.2 Opérations de modification sur les graphes

Nous présentons quelques opérations que nous utiliserons dans les algorithmes manipulant des arbres et des graphes.

— L'opération CreerNoeud($!\mathcal{G}, E$) ajoute un nouveau nœud étiqueté par E aux nœuds de $\mathcal{G}$ et retourne ce nouveau nœud.

— L'opération AjouterEnfant($!\mathcal{G}, n, n'$) ajoute l'arête $\langle n, n'\rangle$ à $\mathcal{G}$, et ne fait rien si cette arête existe déjà.

2.3 Automates

Les automates sont des structures de données caractéristiques de la théorie des langages. D'un point de vue structurel, les automates sont identiques à des graphes étiquetés, et la différence principale est l'utilisation qui en est faite, ainsi que les notations.

Définition 9. Un **automate** est un tuple $\mathcal{A} = \langle Q, Q_0, \Sigma, \delta, F\rangle$ dans lequel :

— Q est un ensemble d'**états**

— $Q_0 \subseteq Q$ est un ensemble d'états dits "**initiaux**"

— Σ est un ensemble appelé l'**alphabet de l'automate**

— $\delta : Q \times \Sigma \to \text{Part}(Q)$ est une **fonction de transition** non-déterministe

— $P : \Sigma^\infty \to \mathbb{B}$ est une propriété qui permet de déterminer si un mot donné est accepté par l'automate.

Définition 10 (Graphe associé à un automate)**.** Soit $\mathcal{A} = \langle Q, Q_0, \Sigma, \delta, F\rangle$ un automate et $\mathcal{G}_\mathcal{A} = \langle \mathcal{N}, \mathcal{E}, \ell_\mathcal{N}, \ell_\mathcal{E}, E_\mathcal{N}, E_\mathcal{E}\rangle$ le graphe étiqueté associé à $\mathcal{A}$. Les conditions suivantes sont satisfaites par $\mathcal{G}_\mathcal{A}$:

— $\mathcal{N} = Q$;

— $\mathcal{E} = \{\langle q_1, q_2\rangle \in Q^2 \mid \exists s \in \Sigma, q_2 \in \delta(q_1, s)\}$;

— $E_\mathcal{N} = \mathbb{B}$;

— $\ell_\mathcal{N}(n) = [n \in Q_0]$;

— $E_\mathcal{E} = \Sigma$;

— $\ell_{\mathcal{E}}(\langle q_1, q_2\rangle) =! s \in \Sigma, q_2 \in \delta(q_1, s)$.

Soient $\mathcal{A}$ un automate, q un état de $\mathcal{A}$, et a un élément de l'alphabet de $\mathcal{A}$. Nous définissons les éléments suivants :

— ETATS($\mathcal{A}$) désigne l'ensemble des états de $\mathcal{A}$

— INITIAUX($\mathcal{A}$) désigne l'ensemble des états initiaux de $\mathcal{A}$

— ALPHABET($\mathcal{A}$) désigne l'alphabet de $\mathcal{A}$

— TRANSITIONS($\mathcal{A}, q, a$) désigne l'ensembles des états calculés par la fonction de transition

$$\text{TRANSITIONS}(\mathcal{A}, q, a) = \delta(q, a)$$

Un automate $\mathcal{A}$ est dit "**fini**" si ETATS($\mathcal{A}$) est fini, et il est dit **déterministe** si pour chaque état $q \in$ ETATS($\mathcal{A}$) et chaque élément $a \in$ ALPHABET($\mathcal{A}$), $|\text{TRANSITIONS}(\mathcal{A}, q, a)| \leqslant 1$.

Nous distinguons les automates sur les mots finis des automates sur les mots infinis.

Définition 11 (Automate fini sur les mots finis)**.** Un **automate fini sur les mots finis** est un automate dont la condition d'acceptation est spécifiée par un sous-ensemble de $F \subseteq Q$, que nous noterons FINAUX($\mathcal{A}$). Soit $\mathcal{A}$ un automate fini sur les mots finis. Un mot $\sigma \in$ ALPHABET$(\mathcal{A})^*$ est **accepté** par $\mathcal{A}$ s'il existe une séquence finie d'états $\xi \in$ ETATS$(\mathcal{A})^{|\sigma|+1}$ telle que les conditions suivantes sont satisfaites :

— $\xi[0] \in$ INITIAUX($\mathcal{A}$)

— $\forall k \in [1 \ldots |\sigma|, \xi[k+1] \in \text{TRANSITIONS}(\mathcal{A}, \xi[k], \sigma[k])$

— $\xi[|\xi|] \in$ FINAUX($\mathcal{A}$)

Avant d'énoncer les résultats principaux sur les automates finis, nous rappelons la définition d'un langage régulier :

Définition 12 (Langage régulier)**.** Soit Σ une alphabet. Les langages réguliers sont obtenus par itération finie des constructions suivantes :

— $\{\varepsilon\}$ (le mot vide) est un langage régulier ;

— Pour tout $s \in \Sigma$, $\{s\}$ est un langage régulier ;

— Pour tous langages réguliers L_1 et L_2, le langage $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$ est un langage régulier ;

— Pour tous langages réguliers L_1 et L_2, $L_1 \cup L_2$ est un langage régulier ;

— Pour tout langage régulier L, $L^* = \bigcup_{k \geqslant 0} L^k$ est un langage régulier, où $L^k = L \cdot L^{k-1}$ pour tout $k > 0$, et $L^0 = \{\varepsilon\}$.

Propriété 1. *Deux résultats principaux de la théorie des langages sur les mots finis sont les suivants, dont les preuves figurent dans la plupart des ouvrages sur le sujet [71] :*

— *Pour tout automate fini non-déterministe sur les mots finis, il existe un automate fini déterministe sur les mots finis qui accepte le même langage ;*

— *L'ensemble des langages reconnus par des automates finis sur les mots finis est exactement équivalent à l'ensemble des langages réguliers.*

En ce qui concerne les automates sur les mots infinis, les conditions mathématiques utilisées pour définir la notion d'acceptation sur les mots infinis se prêtent mal à une application algorithmique directe, car il est algorithmiquement inconcevable de parcourir un mot infini de bout en bout. En revanche, étant donné une représentation finie d'un ensemble de mots infinis, il est parfois possible de déterminer si chacun des mots de cet ensemble est accepté par un automate donné. Les procédures de reconnaissance algorithmique de mots infinis doivent donc nécessairement faire appel à des notions de théorie de la preuve, au moins de façon implicite.

Les applications des automates sur les mots infinis en algorithmique concernent principalement le domaine de la vérification formelle de programmes réactifs, dont le but consiste à améliorer la qualités des programmes informatiques en vérifiant de façon automatisée que certaines propriétés données dans un langage de spécification adéquat sont satisfaites. En ce qui concerne la spécification des programmes, l'idéal serait de pouvoir soumettre le code source du programme tel qu'il est est écrit, mais la complexité syntaxique des langages de programmation conduit naturellement à travailler sur des langages plus simple dédiés à cet objectif. En ce qui concerne la spécification des propriétés, les formalismes les plus utilisés actuellement s'inspirent de la logique temporelle linéaire et de la logique temporelle arborescente.

Il existe une variété de conditions d'acceptation, parmi lesquelles la condition de Büchi, la condition de Büchi généralisée, la condition de Rabin, la condition de Muller, la condition de Street, ou la condition de parité. Ces conditions offrent différents avantages ou inconvénients algorithmiques, mais les versions non-déterministes de ces conditions correspondent toutes aux langages ω-réguliers. Un résumé concis des diverses conditions d'acceptation figurent dans [48], et les complexité de transduction d'une condition d'acceptation vers une autre figurent en partie dans [117].

Une propriété particulière des automates de Büchi est que les variantes déterministes ne reconnaissent qu'un sous-ensemble des langages reconnus par les variantes non-déterministes, ce qui rend difficile le problème de synthèse de l'automate complémentaire [145]. Dans le cas de la logique temporelle linéaire, l'utilisation de la négation de la spécification, qui peut se faire en temps linéaire, permet de rendre la procédure de vérification plus efficace [143].

Nous donnons ici seulement les définitions relatives aux automates de Büchi. Les définitions des autres condition d'acceptations peuvent être aisément trouvées dans la littérature.

Définition 13 (Automate de Büchi)**.** Un **automate de Büchi** est un automate sur les mots infinis $\mathcal{A}$ dont la condition d'acceptation est spécifiée par un ensemble d'états $\text{BUCHI}(\mathcal{A}) \subseteq \text{ETATS}(\mathcal{A})$ dits "**états finaux**". Soit $\mathcal{A}$ un automate de Büchi. Un mot $\sigma \in \text{ALPHABET}(\mathcal{A})^\omega$ est **accepté** par $\mathcal{A}$ s'il existe une séquence infinie d'états $\xi \in \text{ETATS}(\mathcal{A})^\omega$ dans laquelle au moins un des états de $\text{BUCHI}(\mathcal{A})$ apparaît une infinité de fois, c'est-à-dire telle que les conditions suivantes sont satisfaites :

— $\xi[0] \in \text{INITIAUX}(\mathcal{A})$
— $\forall k \in [1 \dots |\sigma|, \xi[k+1] \in \text{TRANSITIONS}(\mathcal{A}, \xi[k], \sigma[k])$
— $\exists q \in \text{BUCHI}(\mathcal{A}), |\{k \in \mathbb{N} \mid \xi[k] = q\}| = \aleph_0$

La synthèse d'un automate associé à une formule de logique temporelle linéaire se fait plus simplement en utilisant une variante de la condition de Büchi.

Définition 14 (Automate de Büchi généralisé)**.** Un **automate de Büchi généralisé** est un automate sur les mots infinis $\mathcal{A}$ dont la condition d'acceptation est spécifiée par un ensemble

d'états BUCHIG($\mathcal{A}$) $\in$ Part(ETATS($\mathcal{A}$))*. Soit $\mathcal{A}$ un automate de Büchi généralisé. Un mot $\sigma \in$ ALPHABET($\mathcal{A}$)$^\omega$ est **accepté** par $\mathcal{A}$ s'il existe une séquence infinie d'états $\xi \in$ ETATS($\mathcal{A}$)$^\omega$ dans laquelle au moins un des états de chaque élément de la séquence BUCHIG($\mathcal{A}$) apparaît une infinité de fois, c'est-à-dire telle que les conditions suivantes sont satisfaites :

— $\xi[0] \in$ INITIAUX($\mathcal{A}$)

— $\forall k \in [1 \ldots |\sigma|, \xi[k+1] \in$ TRANSITIONS$(\mathcal{A}, \xi[k], \sigma[k])$

— $\forall i \in |$BUCHIG$(\mathcal{A})|, \exists q \in$ BUCHIG$(\mathcal{A})[i], |\{k \in \mathbb{N} \mid \xi[k] = q\}| = \aleph_0$

Un automate de Büchi généralisé comportant n états et k ensembles d'états finaux est équivalent à un automate de Büchi comportant $k \cdot n$ états [60]. Le principe consiste à vérifier que les k ensembles d'états finaux ont été traversé en utilisant un compteur modulaire, et de modifier la fonction de transitions en intégrant ce compteur afin que la condition généralisée soit respectée. Nous donnons un exemple d'automate de Büchi généralisé dans le chapitre 4.

2.4 Formules logiques

Une formule logique est une combinaison de symboles d'arités variées avec des constantes, des variables, ou d'autres formules logiques. Les notions relatives aux formules logiques sont définies plus en détail dans le chapitre 4.

Structurellement, une formule logique est un arbre étiqueté $\mathcal{T} = \langle \mathcal{N}, \mathcal{E}, \ell_\mathcal{N}, E_\mathcal{N}$, accompagné d'une relation d'ordre totale sur $\mathcal{N}$.

L'ensemble des nœuds $\mathcal{N}$ est un sous-ensemble arbitraire de $\mathbb{N}$, et l'ensemble des étiquettes $E_\mathcal{N}$ contient les symboles des opérateurs, des variables et des constantes. Pour qu'un arbre corresponde à une formule syntaxiquement valide, il est nécessaire d'imposer des conditions supplémentaires qui dépendent de la syntaxe de la logique considérée.

Les étiquettes permettent d'autoriser différents nœuds à porter des étiquettes identiques, ce qui est nécessaire. La relation d'ordre permet de distinguer la position d'un nœud enfant dans l'ensemble des nœuds enfants, ce qui est nécessaire afin de pouvoir distinguer, par exemple, l'antécédent du conséquent dans une formule de la forme $\phi \to \theta$.

Dans les algorithmes qui doivent manipuler des ensembles de formules, nous introduisons des notations spécifiques pour sélectionner des formules particulières. Soient E un ensemble de formules, φ une formule, et θ une variable libre dans φ. Alors la construction "**Pour** $\varphi \in E$ **faire**" permet d'itérer sur toutes les formules qui ont la même structure que φ par unification, et la notation $\{\theta \mid \varphi \in E\}$ permet de garder seulement les fragments qui sont à l'emplacement de θ dans les formules dont la forme satisfait φ. Ceci ne peut fonctionner correctement qu'à condition que les unifications puissent être effectuées d'une unique façon, hypothèse qui n'est pas vraie en général, mais qui sera vérifiée dans les applications considérées dans la suite.

Exemple 1. Soit $E = \{p \vee q, p \wedge (q \vee r), (p \wedge q) \vee r\}$. Alors

$$\{\theta \mid \theta \vee \psi \in E\} = \{p, p \wedge q\}$$

et

$$\{\psi \to \theta \mid \theta \vee \psi \in E\} = \{q \to p, r \to (p \wedge q)\}$$

Exemple 2. Soit $E = \{p \vee q, p \wedge (q \vee r), (p \wedge q) \vee r\}$. L'exécution de l'algorithme 2.1 sur E produit le résultat suivant :

$$\texttt{ExempleBoucle}(E) = \{p, q, r, p \wedge q\}$$

Algorithme 2.1 Exemple d'itération avec unification

```
1: procédure ExempleBoucle(E : Ens)
2:     R = ∅
3:     pour chaque ψ ∨ θ ∈ E faire
4:         Ajouter(R, ψ)
5:         Ajouter(R, θ)
6:     fin pour
7:     renvoyer R
   fin procédure
```

2.5 Considérations algorithmiques

Il n'existe à l'heure actuelle aucune standardisation de la notion d'algorithme qui soit complètement définie est standardisée. Les notations que nous avons introduites jusqu'à présent permettent d'utiliser certaines structures de données mathématiques comme des objets, mais en utilisant des fonctions qui ressemblent plus aux fonctions mathématiques habituelles.

La différence principale entre les notations mathématiques et les notations algorithmiques est qu'en mathématiques, les objets sont supposés pré-existants au discours, et que le mathématician ne fait que référencer ces objets par des représentations diverses, alors qu'en algorithmique, les objets considérés sont construits et modifiés au cours du temps. Ceci pose un problème de notation, car il est parfois beaucoup plus naturel de modifier un objet que d'en effectuer une copie. Cependant, le support notationnel est pratiquement insexistant, ce qui fait qu'il est souvent laissé au lecteur la nécessité de devenu quels objeets sont modifiés par certaines fonctions, comme si cela allait de soit. Nous utiliserons le point d'exclamation (‘ !’) pour dénoter les objets qui seront modifiés par une fonction. En général, un unique objet est modifié, en utilisant les informations fournies par les autres paramètres de la fonction.

Ainsi, une fonction dont la signature est $\texttt{f}(!x, y, z)$ est une fonction qui modifiera l'objet x en exploitant les objets y et z. Les autres notations qui seront utilisées sont similaires à celles d'un langage impératif classique, et il n'est pas nécessaire de les détailler ici.

Chapitre 3

Programmation par contraintes

Sommaire

La *programmation par contraintes* (PPC) est un paradigme de programmation déclaratif dans lequel un problème combinatoire est décrit par un ensemble de contraintes et est résolu automatiquement par un solveur de contraintes. Dans ce chapitre, nous décrivons la programmation par contraintes dans ses grandes lignes, car depuis la création de la conférence CP (*Principles and Practice of Constraint Programming*) [97] et l'ouvrage de référence *Foundations of Constraint Programming* [135], ce domaine a énormément évolué. Le *Handbook Of Constraint Programming* [114] présente un état de l'art qui couvre de nombreux aspects fondamentaux du domaine.

3.1 Introduction

Les contraintes constituent l'interface utilisateur de la PPC. L'utilisateur d'un solveur de contraintes a seulement besoin de connaître les contraintes qui sont à sa disposition, et de déterminer comment exprimer le problème qu'il a à résoudre par une conjonction de contraintes.

Les méthodes algorithmiques les plus utilisées pour la résolution de problèmes de satisfaction de contraintes sont basées sur la recherche en profondeur, qui consiste dans sa présentation la plus simple à essayer différentes instanciations des variables jusqu'à en trouver une qui soit une solution. Cette méthode naïve est fondamentalement inefficace, car de complexité exponentielle en le produit des tailles des domaines des variables.

Les notions de consistances locales permettent d'améliorer les performances en limitant la taille de l'espace de recherche exploré, par le biais de déductions basées sur les sémantiques des contraintes. Ces déductions sont généralement implémentées dans des propagateurs, qui sont des algorithmes spécifiques à chaque contrainte.

Le problème de satisfaction de contraintes étant NP-complet, il est illusoire d'espérer pouvoir résoudre toutes les instances de ce problème en temps polynomial, même en utilisant des procédures d'applications de consistances aussi évoluée fussent-elles. Pour certaines familles d'instances, il est possible de définir des heuristiques qui permettent de rendre la recherche sensibleemnt plus efficace, voir polynomiale [35]. Ces heuristiques sont des combinaisons de règles concernant le choix des variables à modifier, le choix des valeurs à utiliser, le choix des contraintes à propager, ou le choix des niveaux de consistances à utiliser pour chaque contrainte. Les stratégies de sélections aléatoires, augmentées de mécanismes de mémorisation et de redémarrage, permettent d'explorer des parties diverses d'un espace de recherche de plus en plus petit, ce qui augmente la probabilité de trouver une solution, et permet de limiter l'influence de mauvais choix effectués par les heuristiques.

L'approche de la programmation par contraintes que nous considérons correspond à celle utilisée dans le solveur de contraintes Gecode [57], qui est une librairie implémentée dans le langage impératif C++, et dont les principes sont discutés dans la thèse [131].

Le plan de ce chapitre est le suivant. Dans la section 3.2, nous donnons les définitions relatives au problème de satisfaction de contraintes. Nous décrivons les algorithmes de recherche en profondeur dans la section 3.3, ainsi que ses variations les plus courantes, puis nous décrivons les notions de consistance et de propagation dans la section 3.4. Nous décrivons brièvement la problématique de l'optimisation sous contraintes dans la section 3.5, et nous décrivons un exemple de problème de satisfaction de contraintes dans la section 3.6. Enfin, certains formalismes déclaratifs apparentés à la programmation par contraintes sont évoqués dans la section 3.7.

3.2 Définitions

Le concept central à la programmation par contraintes est la notion de problème de satisfaction de contraintes.

Définition 15. Un **problème de satisfaction de contraintes** (CSP) est un tuple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ dans lequel :

— $\mathcal{X}$ est un ensemble fini de **variables** ;

— $\mathcal{D}$ est une fonction qui associe à chaque variable $X \in \mathcal{X}$ son **domaine** $\mathcal{D}(X)$;

— $\mathcal{C}$ est un ensemble fini de contraintes d'arités finies sur $\mathcal{X}$.

Une *variable* est simplement un identifiant qui peut être utilisé dans la spécification des contraintes, et auquel est associé un *domaine*. Dans le cadre de la PPC, le domaine des variables est en général supposé de cardinalité finie, donc discret.

Afin de simplifier les notations et sans perdre en généralité, nous supposons que les domaines des variables sont des sous-ensembles finis de nombres entiers, c'est-à-dire que pour toute variable $X \in \mathcal{X}$, le domaine $\mathcal{D}(X)$ de X est un sous-ensemble de $\mathbb{Z}$. Cette hypothèse est naturelle si les contraintes sont des contraintes arithmétiques sur les nombres entiers, mais il existe d'autres types de contraintes, et surtout des contraintes sur d'autres types de variables, telles que les nombres flottants [92], les ensembles [61], les arbres [128] ou les graphes [41]. Il est préférable de considérer la programmation par contraintes sur ces types de variables comme des adaptatations des techniques de la programmation par contraintes sur les variables entières, et de considérer que la programmation par contraintes sans mention explicite du type de variable considéré se réfère à la programmation par contraintes sur les variables entières.

Une contrainte est une spécification d'un sous-ensemble du produit cartésien des domaines d'un sous-ensemble de variables, qui correspond à l'ensemble des modèles d'une formule d'une logique appropriée [30].

Définition 16 (Contrainte)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP. Une **contrainte** $c \in \mathcal{C}$ est un tuple $\langle \mathrm{var}(c), \varphi(V), \mu \rangle$ dans lequel :

— $\mathrm{var}(c) \subseteq \mathcal{X}$ est un sous-ensemble de variables appelé la **portée** de c ;

— φ est une formule logique sur un ensemble V de variables libres dans φ, à valeurs dans $\mathbb{Z}$;

— $\mu : V \to \mathrm{var}(c)$ est une fonction surjective qui associe à chaque variable de V une variable de $\mathrm{var}(c)$.

Une contrainte globale est l'extension de la notion de contrainte à un nombre de variables arbitraire. Il ne semble pas qu'un langage de spécification universel pour les contraintes globales ait été défini, mais l'existence d'un tel langage n'est pas nécessaire, dans la mesure où la sémantique des contraintes globales est encodée manuellement dans les solveurs de contraintes. Une définition générique de la notion de contrainte globale nécessite d'utiliser un système de type en paramètre. Bien que les prémisses d'un tel système soient présentes dans le *Global Constraint Catalog*, [15], le fait que la sémantique de nombreuses contraintes y soient décrites seulement en langue naturelle est un signe de l'absence de système logique pour la description de la sémantique des contraintes en général.

Définition 17 (Contrainte globale)**.** Une **contrainte globale** est une contrainte dont la spécification s'applique à un nombre variable de variables. Elle est caractérisée par une spécification, sous la forme d'une formule logique ou d'un algorithme permettant d'établir si la constrainte est satisfaite, et par le type des arguments que l'utilisateur doit fournir lors de l'utilisation de la contrainte.

Une contrainte globale représente une famille infinie de contraintes, qui sont définies par des formules similaires, et pour lesquelles les mêmes algorithmes peuvent être utilisés.

Exemple 3 (Contrainte `atmost`)**.** La contrainte globale `atmost` spécifie qu'une valeur v donnée doit apparaître au plus k fois dans un ensemble de variables X, ce qui correspond à la formule de logique du premier ordre φ suivante :

$$\varphi(I) = |\{x \in X \mid I(x) = v\}| \leqslant k$$

où I est une instanciation des variables de X (cf. définition suivante). L'instance de `atmost` C qui porte sur le tuple de variables $\langle X, Y, X\rangle$ avec $v = 1$ et $k = 2$ admet la définition du premier ordre suivante :

— $\text{var}(C) = \{X, Y\}$;
— $\varphi = (x_1 \neq 1 \vee x_2 \neq 1 \vee x_3 \neq 1)$;
— $\mu = \{x_1 \mapsto X, x_2 \mapsto Y, x_3 \mapsto X\}$.

Bien que l'algorithme de propagation soit identique pour toutes les instances des contraintes `atmost`, une spécification du première ordre doit être spécifique à une instance donnée. La caractérisation du type de logique utilisée pour décrire la contrainte `atmost` dans sa globalité est laissée au lecteur.

La notion d'instanciation est fondamentale en programmation par contraintes. Il s'agit de la même notion que la notion d'interprétation utilisée en logique (Chapitre 4), mais la terminologie associée est particulièrement développée en ce qui concerne la programmation par contraintes.

Définition 18 (Instanciations)**.** Une **instanciation** I d'un ensemble de variables $X \subseteq \mathcal{X}$ de domaines $\mathcal{D} : \mathcal{X} \to \mathbb{Z}$ est une fonction totale $I : X \to \mathbb{Z}$ qui associe à chaque variable $x \in X$ une valeur $I(x) \in \mathcal{D}(x)$. La portée d'une instanciation I est notée $\text{var}(I)$. Une **instanciation complète** est une instanciation qui porte sur l'ensemble des variables, c'est-à-dire une instanciation I pour laquelle $\text{var}(I) = \mathcal{X}$. Une **instanciation partielle** est une instanciation qui n'est pas complète. Une **instanciation singleton** est une instanciation qui porte sur une seule variable $x \in \mathcal{X}$, c'est-à-dire une instanciation I pour laquelle $\text{var}(I) = \{x\}$. Une **valeur** d'une instanciation I est une paire (x, v) avec $x \in \text{var}(I)$ et $v \in \mathcal{D}(x)$. Une variable $x \in \mathcal{X}$ est **instanciée** dans une instanciation I si $x \in \text{var}(I)$, et est dite "**non instanciée**" sinon. L'**ensemble des instanciations** sur des variables $X \subseteq \mathcal{X}$ est noté $\mathcal{I}_{\mathcal{D}}(X)$, ou bien simplement $\mathcal{I}(X)$ lorsqu'il n'y a pas d'ambiguïté.

Diverses opérations peuvent être appliquées sur des instanciations, en particulier les opérations de projection, d'extension, et d'union.

Définition 19 (Projection et extension d'instanciations)**.** La **projection** d'une instanciation I sur un ensemble de variable $Y \subseteq \text{var}(I)$ est l'unique instanciation $I{\downarrow}X : X \to \mathbb{Z}$ telle que pour chaque variable $y \in Y$, $(I{\downarrow}X)(y) = I(y)$. Une **extension** d'une instanciation partielle I à des variables $Y \subseteq \mathcal{X}\backslash\text{var}(I)$ est une instanciation I' de domaine $\text{var}(I') = \text{var}(I) \cup Y$ qui est telle que $(I'){\downarrow}\text{var}(I) = I$. L'ensemble des extensions d'une instanciation I à un ensemble de variables Y est noté $I{\uparrow}Y$.

Définition 20 (Union d'instanciations)**.** Soient I et I' deux instanciations telles que pour chaque variable $x \in \text{var}(I) \cap \text{var}(I')$, $I(x) = I'(x)$. Alors l'**union** de I et I' est l'unique instanciation $I \cup I'$ pour laquelle $\text{var}(I \cup I') = \text{var}(I) \cup \text{var}(I')$ et telle que $(I \cup I')(x) = I(x)$ si $x \in \text{var}(I)$ et $(I \cup I')(x) = I'(x)$ si $x \in \text{var}(I')$.

Une propriété élémentaire mais importante des ensembles d'instanciations des variables d'un CSP est que leurs cardinalités sont finies.

Propriété 2 (Nombre d'instanciations)**.** *Soient $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP, et $X \subseteq \mathcal{X}$ un sous-ensemble des variables de $\mathcal{P}$. Alors le nombre d'instanciations sur X est donné par :*

$$|\mathcal{I}(X)| = \prod_{x \in X} |\mathcal{D}(x)|$$

Nous pouvons maintenant définir la notion de solution d'une contrainte et de solution d'un CSP en terme d'instanciations.

Observation 1 (Instanciation et modèle)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP et $c = \langle \text{var}(c), \varphi, \mu \rangle$ une contrainte de $\mathcal{P}$. Noter que si $I \in \mathcal{I}(\text{var}(c))$ est une instanciation des variables de c, alors $(\mu \circ I)$ est une interprétation des variables de φ, où '$\circ$' dénote la composition de fonctions.

$$\forall v \in var(\varphi), (\mu \circ I)(v) = I(\mu(v))$$

Définition 21 (Solution d'une contrainte)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP et $c = \langle \text{var}(c), \varphi, \mu \rangle$ une contrainte de $\mathcal{P}$. Une instanciation $I \in \mathcal{I}(\text{var}(c))$ est une **solution de** c si $(\mu \circ I) \models \varphi$, c'est-à-dire si $\mu \circ I$ est un modèle de φ. L'ensemble des solutions de c est noté $\text{Sol}(c)$.

Définition 22 (Solution d'un CSP)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP. Une instanciation $I \in \mathcal{I}(\mathcal{X})$ est une **solution de** $\mathcal{P}$ si pour toute contrainte $c \in \mathcal{C}$, $(I \downarrow \text{var}(c)) \in \text{Sol}(c)$. L'ensemble des solutions de $\mathcal{P}$ est noté $\text{Sol}(\mathcal{P})$.

Définition 23 (Satisfaction)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP. Une contrainte $c \in \mathcal{C}$ est **satisfaisable** s'il existe une instanciation $I \in \mathcal{I}(\text{var}(c))$ qui est une solution de c. Un CSP $\mathcal{P}$ est **satisfaisable** s'il existe une solution de $\mathcal{P}$, c'est-à-dire si $\text{Sol}(\mathcal{P}) \neq \emptyset$. Une contrainte et un CSP sont **insatisfaisable** s'ils ne sont pas satisfaisables.

3.3 Recherche en profondeur

Le mécanisme de recherche de solution s'explique le plus simplement avec la notion de sous-problème [18]. Un sous-problème d'un CSP $\mathcal{P}$ est un CSP $\mathcal{P}'$ similaire à $\mathcal{P}$ qui ne possède aucune solution qui ne soit pas une solution de $\mathcal{P}$.

Définition 24 (Sous-problème)**.** Soient $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ et $\mathcal{P}' = \langle \mathcal{X}', \mathcal{D}', \mathcal{C}' \rangle$ deux CSPs. De façon très générale, $\mathcal{P}'$ est un **sous-problème** de $\mathcal{P}$, noté $\mathcal{P}' \prec \mathcal{P}$, si $\mathcal{X} = \mathcal{X}'$ et $\text{Sol}(\mathcal{P}') \subset \text{Sol}(\mathcal{P})$.

Il existe deux façons de restreindre l'ensemble des solutions d'un CSP. La plus fréquente consiste à restreindre les domaines des variables.

Définition 25 (Sous-problème basé domaine)**.** Soient $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ et $\mathcal{P}' = \langle \mathcal{X}', \mathcal{D}', \mathcal{C}' \rangle$ deux CSP. $\mathcal{P}'$ est un **sous-problème basé domaine** de $\mathcal{P}$, noté $\mathcal{P}' \prec_D \mathcal{P}$ si et seulement si les domaines des variables sont restreints, c'est-à-dire si les conditions suivantes sont satisfaites :

- $\mathcal{P}' \prec \mathcal{P}$;
- pour toute variable $x \in \mathcal{X}$, $\mathcal{D}'(x) \subseteq \mathcal{D}(x)$;
- il existe au moins une variable $x \in \mathcal{X}$ telle que $\mathcal{D}'(x) \subset \mathcal{D}(x)$;

— $\mathcal{C} = \mathcal{C}'$ [1].

L'autre possibilité pour restreindre les solutions d'un CSP consiste à ajouter des contraintes, ou à restreindre certaines contraintes en supprimant certaines instanciations de leurs ensembles d'instanciations.

Définition 26 (Sous-problème basé contrainte)**.** Soient $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ et $\mathcal{P}' = \langle \mathcal{X}', \mathcal{D}', \mathcal{C}' \rangle$ deux CSP. $\mathcal{P}'$ est un **sous-problème basé contrainte** de $\mathcal{P}$, noté $\mathcal{P}' \prec_C \mathcal{P}$ s'il existe des contraintes supplémentaires dans $\mathcal{P}'$ ou si certaines contraintes sont restreintes, c'est-à-dire si les conditions suivantes sont satisfaites :

— $\mathcal{P}' \prec \mathcal{P}$

— $\mathcal{D} = \mathcal{D}'$

— $\mathcal{C} \neq \mathcal{C}'$

— Pour toute contrainte $c = \langle \text{var}(c), \varphi, \mu \rangle \in \mathcal{C}$, soit il existe une contrainte $c' = \langle \text{var}(c'), \varphi', \mu' \rangle \in \mathcal{C}'$ telle que $c = c'$, soit il existe une contrainte $c'' = \langle \text{var}(c''), \varphi'', \mu'' \rangle \in \mathcal{C}''$ telle que $\varphi'' = \varphi \wedge \theta$ où θ spécifie des contraintes additionnelles pour c''.

Ces conditions imposent que l'ensemble de contraintes de $\mathcal{P}$ est modifié, soit par l'extension d'une contrainte existante, soit par l'ajout de contraintes.

La modification de contrainte n'est techniquement réalisable que pour des contraintes dont les instanciations peuvent être manipulées explicitement. Les techniques de mémorisation [119], ainsi que les contraintes utilisant des notions de consistance de chemin nécessitent de telles représentations. Comme Gecode est construit sur la notion de propagateur de contraintes, la notion de sous-problème basé domaine est la plus importante dans ce contexte.

3.3.1 Arbre de recherche

Le mécanisme de recherche en profondeur conduit à la construction d'un arbre de recherche. Les algorithmes de recherche en profondeur les plus communément utilisés réalisent principalement des opérations de modification des domaines des variables pour générer des sous-problèmes. Nous donnons d'abord quelques définitions, puis nous décrivons les trois possibilités les plus fréquentes pour effectuer les modification de domaines. Un algorithme de recherche en profondeur est donné en fin de section.

Différentes méthodes de recherche en profondeur produisent des arbres de recherche différents, mais ceux-ci doivent tous satisfaire la définiton suivante.

Définition 27 (Arbre de recherche)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP. Un **arbre de recherche** pour $\mathcal{P}$ est un arbre non-étiqueté $\mathcal{T} = \langle N, p \rangle$ dans lequel les nœuds sont des CSPs, et qui satisfait les conditions suivantes :

1. La racine de $\mathcal{T}$ est le CSP lui-même
   $\text{RACINE}(\mathcal{T}) = \mathcal{P}$;
2. Tout nœud enfant est un sous-problème de son nœud parent
   $\forall n \in \text{NOEUDS}(\mathcal{T}) \backslash \text{RACINE}(\mathcal{T}), n \prec \text{PARENT}(n)$;

1. La notion d'égalité utilisée est la notion d'égalité syntaxique, c'est-à-dire que les même contraintes sont utilisées avec les même formules. Une telle égalité est décidable.

3. L'ensemble des solutions d'un nœud est identique à l'union des ensembles de solutions de ses nœuds enfants
   $\forall n \in \text{NOEUDS}(\mathcal{T}) \backslash \text{FEUILLES}(\mathcal{T}), \bigcup_{n' \in \text{ENFANTS}(n)} \text{Sol}(n') = \text{Sol}(n)$;
4. Les nœuds feuille de l'arbre de recherche sont soit des nœuds insatisfaisables, soit des solutions
   $\forall n \in \text{FEUILLES}(\mathcal{T}), |\text{Sol}(n)| \leqslant 1$.

3.3.2 Modification de domaines

Pour générer des sous-problèmes, il est possible de modifier les domaines de plusieurs variables à la fois, mais les méthodes habituelles effectuent en général la modification d'une seule variable. Les possibilités de modification du domaine d'une unique variable sont elles-mêmes multiples.

Définition 28 (Modification de domaines)**.** Soient $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP et $x \in \mathcal{X}$ une variable de $\mathcal{P}$. Trois méthodes pour générer des sous-problèmes sont les suivantes [136]. Nous notons $e(\mathcal{P})$ l'ensemble de ces sous-problèmes.

— **Instanciation.** Le principe consiste à générer autant de sous-problèmes qu'il y a de valeurs dans $\mathcal{D}(x)$. Dans ce cas, $e(\mathcal{P}) = \{\mathcal{P}_v \mid v \in \mathcal{D}(v)\}$, avec $\mathcal{P}_v = \{\mathcal{X}_v, \mathcal{D}_v, \mathcal{C}_v\}$, $\mathcal{D}_v(x) = \{v\}$, $\mathcal{X}_v = \mathcal{X}$, $\mathcal{C}_v = \mathcal{C}$, et $\mathcal{D}_v(y) = \mathcal{D}(y)$ pour $y \neq x$.

— **Dichotomie.** Le principe consiste à choisir une valeur $v \in \mathcal{D}(x)$, et à générer deux sous-problèmes $e(\mathcal{P}) = \{\mathcal{P}_v = \langle \mathcal{X}_v, \mathcal{D}_v, \mathcal{C}_v \rangle, \mathcal{P}_{\neg v} = \langle \mathcal{X}_{\neg v}, \mathcal{D}_{\neg v}, \mathcal{C}_{\neg v} \}\rangle$ pour lesquels :
   — $\mathcal{X}_v = \mathcal{X}$, $\mathcal{C}_v = \mathcal{C}$, $\mathcal{D}_v(x) = \{v\}$ et $\mathcal{D}_v(y) = \mathcal{D}(y)$ pour $y \in \mathcal{X} \backslash \{x\}$.
   — $\mathcal{X}_{\neg v} = \mathcal{X}$, $\mathcal{C}_{\neg v} = \mathcal{C}$, $\mathcal{D}_{\neg v}(x) = \mathcal{X} \backslash \{v\}$ et $\mathcal{D}_{\neg v}(y) = \mathcal{D}(y)$ pour $y \in \mathcal{X} \backslash \{x\}$.

— **Disjonction.** Dans cette méthode plus générique, deux sous-ensemble $V_1 \subseteq \mathcal{D}(x)$ et $V_2 \subseteq \mathcal{D}(x)$ sont définis, pour lesquels $V_1 \cap V_2 = \emptyset$ et $V_1 \cup V_2 = \mathcal{D}(V)$. Les deux sous-problèmes correspondants sont $e(\mathcal{P}) = \{\mathcal{P}_1 = \langle \mathcal{X}_1, \mathcal{D}_1, \mathcal{C}_1 \rangle, \mathcal{P}_2 = \langle \mathcal{X}_2, \mathcal{D}_2, \mathcal{C}_2 \}\rangle$, avec, pour $i \in \{1, 2\}$, $\mathcal{X}_i = \mathcal{X}$, $\mathcal{C}_i = \mathcal{C}$, $\mathcal{D}_i(x) = V_i$, et $\mathcal{D}_i(y) = \mathcal{D}(y)$ pour $y \in \mathcal{X} \backslash \{x\}$.

Ces méthodes peuvent être combinées de diverses façons, mais sans l'usage de techniques de propagations, elles produisent toutes des arbre de recherche dont les nombres de nœuds feuille sont identiques. Les avantages, inconvénients et méthodes alternatives sont développées dans [136]. Nous les mentionnant ici seulement afin de motiver l'abstraction qui est faite dans les algorithmes qui suivent.

3.3.3 Algorithme

Nous présentons dans cette section un algorithme générique de recherche en profondeur (Algorithme 3.1). Soit $\mathcal{P}_0 = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ le CSP à résoudre. Nous supposons l'existence d'une fonction '`Couper`' qui prend en entrée un problème $\mathcal{P}_w$ et produit en sortie n_w sous-problèmes $\mathcal{P}_{w,0}$, $\mathcal{P}_{w,1}$, …, $\mathcal{P}_{w,n_w}$, qui sont tels que

$$\bigcup_{k \in [1 \ldots n_w]} \text{Sol}(\mathcal{P}_{w,k}) = \text{Sol}(\mathcal{P}_w)$$

Une restriction supplémentaire relativement forte qui garantie la terminaison est de s'assurer que tous les sous-problème comportent moins d'instanciations que le problème parent, c'est-à-dire que :

$$\forall k \in [1 \ldots n_w], \mathcal{I}(\text{var}(\mathcal{P}_{w,k})) \subset \mathcal{I}(\text{var}(\mathcal{P}_w))$$

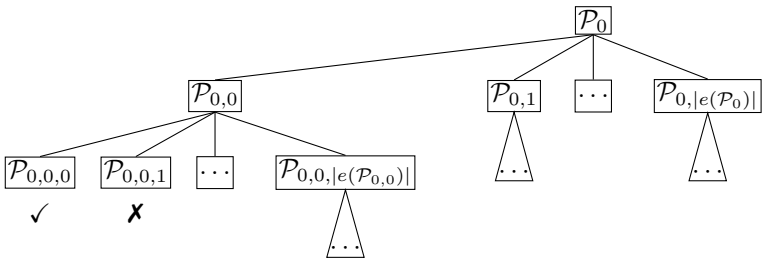

FIGURE 3.1 – Exemple d'arbre de recherche

La fonction principale est la fonction '`Résoudre`', qui utilise la fonction '`Couper`' pour générer des sous-problèmes, puis s'appelle récursivement sur chacun de ces sous-problèmes.

Le comptage de solution d'un CSP est un problème difficile. À la ligne 3 de l'algorithme 3.1, nous comptons les solutions d'un CSP dans lequel les domaines des variables sont au plus de taille unité, ce qui rend le comptage de solutions réalisables dans ce cas extrêmement limité. Le CSP contient alors au plus une solution, et l'écriture utilisée permet d'exprimer concisément les trois possibilités suivantes :

— les domaines de toutes les variables sont des singletons, et le CSP représente une instanciation unique, qui est une solution ;

— les domaines de toutes les variables sont des singletons, et le CSP représente une instanciation unique, qui n'est pas une solution ;

— au moins une des variables a un domaine vide, auquel cas le CSP n'a trivialement pas de solutions.

Algorithme 3.1 Recherche en profondeur simple

```
1: Fonction Résoudre(P_w = ⟨X_w, D_w, C_w⟩ : CSP)
2:   si ∀x ∈ X_w, |D_w| ⩽ 1 alors
3:     si |Sol(P_w)| = 1 alors
4:       afficher Sol(P_w)
5:     fin si
6:   sinon
7:     pour chaque P_{w,k} = ⟨X_{w,k}, D_{w,k}, C_{w,k}⟩ ∈ Couper(P_w) faire
8:       Résoudre(P_{w,k})
9:     fin pour
10:  fin si
11: Fin fonction
```

Nous illustrons l'arbre de recherche correspondant $\mathcal{T}(\mathcal{P}_0)$ dans la figure 3.1. Dans cette example, le CSP $\mathcal{P}_{0,0,0}$ est supposé être une instanciation qui constitue une solution de $\mathcal{P}_0$, et $\mathcal{P}_{0,0,1}$ est supposée être une instanciation qui n'est pas une solution. En ce qui concerne les sous-problèmes $\mathcal{P}_{0,0,|e(\mathcal{P}_{0,0})|}$, $\mathcal{P}_{0,1}$, $\mathcal{P}_{0,|e(\mathcal{P}_0)|}$, l'exécution de l'algorithme 3.1 est laissée non spécifiée, et il en va de même pour les sous problèmes $\mathcal{P}_{0,k}$ avec $1 < k < |e(\mathcal{P}_0)|$ et $\mathcal{P}_{0,0,k}$ avec $1 < k < |e(\mathcal{P}_{0,0})|$. Les CSPs eux-mêmes, ainsi que les choix effectués par l'algorithme, sont également laissés non spécifiées dans cet exemple.

L'algorithme que nous décrivons ici est une version très simplifiée qui omet des optimisations évidentes. Par exemple, si une contrainte dont la portée est incluse dans l'ensemble des variables

instanciées n'est pas satisfaite, alors il est possible d'arrêter immédiatement la recherche, car le sous-espace correspondant ne contient aucune solution. Nous omettons cette optimisation pour plusieurs raisons. D'une part, la détection des contraintes à évaluer et l'évaluation proprement dite de ces contraintes a engendre coût, et ce procédé lui même peut être le sujet d'un certain nombre d'optimisations. Enfin, ceci constitue déjà une forme très simple de propagation, dont la généralisation complète nécessite la notion de consistance et de propagateurs qui sont décrites dans la section suivante.

3.4 Consistences locales et propagateurs

Les opérations de modifications effectuées par l'algorithme 3.1 sont tout à fait arbitraires. Les propriétés de consistances locales sont des propriétés qui permettent de CSPs qui permettent de déterminer des modifications particulières qui sont plus intéressantes que les autres.

Définition 29 (Consistance locale)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP. Une **propriété de consistance locale** Φ est une propriété qui caractérise des conditions nécessaires pour que des instanciations partielles de $\mathcal{X}$ soient extensibles à des solutions. $\mathcal{P}$ est dit Φ **consistant** si toutes les instanciations partielles des variables de $\mathcal{P}$ satisfont Φ. Dans le cas contraire, $\mathcal{P}$ est dit **Φ-inconsistant**.

Si un CSP $\mathcal{P}$ est Φ-inconsistant pour une propriété de consistance locale Φ, alors il existe un ensemble d'instanciations partielles des variables de $\mathcal{P}$ qui ne peuvent pas être étendues à une solution, ce qui implique qu'il existe nécessairement un sous-problème de $\mathcal{P}$ qui est Φ-consistant. L'objectif est alors, étant données des contraintes et une notion de consistance, de déterminer un algorithme capable de calculer efficacement un sous-problème Φ consistant de Φ.

Bien qu'il existe des notions de consistances qui mettent en jeu plusieurs contraintes, les propriétés les plus utilisées en pratiques concernent une seule contrainte. Pour ces notions, il est possible d'associer un algorithme de propagation à chaque contrainte indépendamment des autres, ce qui permet de paramétrer très finement les niveaux de consistances qui seront utilisés en fonction des instances des contraintes.

Définition 30 (Consistance locale de contrainte)**.** Soient $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP, c une contrainte de $\mathcal{P}$ et Φ une propriété de consistance locale. $\mathcal{P}$ est **Φ-consistant pour** c si le CSP $\mathcal{P}' = \langle \mathcal{X}, \mathcal{D}, \{c\} \rangle$ est Φ-consistant.

Les notions de consistances et les algorithmes associés ont initialement été définis pour des contraintes tabulaires. Pour les autres types de contraintes, ces algorithmes sont souvent inutilisables, car d'une complexité prohibitive. Une contrainte globale nécessite donc de créer un algorithme dédié qui sera capable d'appliquer au mieux la notion de consistance désirée en exploitant complètement la sémantique de la contrainte.

Définition 31 (Propagateur)**.** Soit c une contrainte et Φ une propriété de consistance locale. Un **propagateur** pour c est un algorithme qui à tout CSP $\mathcal{P}$ Φ-inconsistant pour c, calcule un sous-problème de $\mathcal{P}$ qui est Φ-consistant pour c.

Nous présentons seulement l'arc consistance génélisée et les consistances aux bornes, qui figurent parmis les propriétés de consistances les plus utilisées en pratique. Une présentation synthétique d'un grand nombre de propriétés de consistances locales figure dans [18].

3.4.1 Consistance d'arc

Les propriétés de consistance d'arc (AC) et de consistance d'arc généralisée (GAC) sont identiques, à ceci près que la première notion a été définie spécifiquement pour les CSPs binaires tandis que la seconde s'applique aux CSPs en général. Les algorithmes de mise en œuvre de l'arc consistance binaire se généralisent facilement aux CSPs généraux, mais alors que la complexité optimale est polynomiale en temps pour les contraintes binaires [95], elle devient exponentielle lorsqu'elle est appliquée à des contraintes d'arités arbitraires [96], car la complexité pour les contraintes tabulaires est exponentielle en l'arité des contraintes. Ceci n'est pas vrai en général pour les contraintes globale, car la sémantique de ces contraintes permet souvent de définir des algorithmes plus performants.

Définition 32 (Consistance d'arc généralisée (GAC))**.** Un CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ est **GAC-consistant** si pour tout instanciation singleton I telle que $\text{var}(I) = \{x\}$ et $I(x) = v$, avec $x \in \mathcal{X}$ et $v \in \mathcal{D}(x)$, et pour toute contrainte $c \in \mathcal{C}$ telle que $x \in \text{var}(c)$, il existe une instanciation $I' \in \mathcal{I}(c)$ telle que $I'(x) = I(x)$.

Définition 33 (Support)**.** Soient $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP, $c \in \mathcal{C}$ une contrainte de $\mathcal{P}$, $x \in \text{var}(c)$ une variable de c et $v \in \mathcal{D}(x)$ une valeur pour x. Un **support** de l'instancation singleton $I(x) = v$ (avec $\text{var}(I) = \{x\}$) pour c est une instanciation $I' \in \mathcal{I}(c)$ telle que $I'(x) = I(x)$.

Propriété 3 (Absence de support)**.** *S'il existe une instanciation singleton I de portée* $\text{var}(I) = \{x\}$ *avec $x \in \mathcal{X}$ et $I(x) = v$ avec $v \in \mathcal{D}(x)$ et une contrainte $c \in \mathcal{C}$ telle que $x \in \text{var}(c)$ mais qu'il n'existe aucune instanciation $I' \in \mathcal{I}(c)$ telle que $I'(x) = I(x)$, alors l'instancation singleton ne peut être étendue à aucune solution*

La propriété 3 implique que la valeur v peut être supprimée du domaine de x s'il existe une constrainte pour laquelle l'instanciation singleton $\{x : v\}$ n'a pas de support. Cette observation est à la base de l'algorithme qui permet d'établir l'arc consistance. La modification du domaine d'une variable implique la modification implicite de toutes les contraintes dont la portée contient cette variable, ce qui génère un effet de propagation en chaîne de l'arc consistance.

3.4.2 Consistances aux bornes

Les notions de consistances aux bornes sont naturelles pour les contraintes arithmétiques. Étant données des variables dont les domaines sont ordonnés, il s'agit de s'intéresser seulement à la borne des domaines des variables, plutôt qu'aux domaines dans leur intégralité.

Notation 1. Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP. Nous notons $\mathcal{D}^\flat$ la fonction qui associe à toute variable $x \in \mathcal{X}$ le domaine $\mathcal{D}^\flat(x) = [\min(\mathcal{D}(x)), \max(\mathcal{D}(x))]$.

Définition 34 (Consistance aux bornes)**.** Soit $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un CSP. Une contrainte $c \in \mathcal{C}$ de $\mathcal{P}$ est **borne-consistante** si et seulement si :

$$\forall x \in \text{var}(c), \exists I \in \mathcal{I}_{\mathcal{D}^\flat}(\text{var}(c)), I(x) = (\min(\mathcal{D}(x)) \vee \max(\mathcal{D}(x)))$$

La consistance aux bornes n'est pas très intéressante pour les contraintes quelconques, car le problème de décision pour des contraintes binaires peut nécessiter un temps exponentiel, alors que la consistance d'arc nécessite seulement un temps polynomial pour ces contraintes. Cependant, pour certaines contraintes, il existe parfois des algorithmes de consistances aux bornes intéressants.

3.4.3 Algorithme

Nous présentons un algorithme générique pour la résolution du problème de satisfaction de contraintes par recherche en profondeur avec propagation. Cette algorithme est similaire à l'algorithme 3.1. Pour chaque contrainte, nous supposons qu'un propagateur $\mathfrak{P}$ lui est associé. Le prédicat `Propagateurs` renvoit l'ensemble des propagateurs pour lesquels une propagation est nécessaire. Un tel ensemble peut-être implémenté par une file de priorités. Une architecture de ce type est présentée dans [131].

Algorithme 3.2 Recherche en profondeur avec propagation

1: **Fonction** `Résoudre`$(\mathcal{P}_w = \langle \mathcal{X}_w, \mathcal{D}_w, \mathcal{C}_w \rangle$: `CSP`$)$
2: **si** $\forall x \in \mathcal{X}_w, |\mathcal{D}_w| \leqslant 1$ **alors**
3: **si** $|\mathrm{Sol}(\mathcal{P}_w)| = 1$ **alors**
4: **afficher** $\mathrm{Sol}(\mathcal{P}_w)$
5: **fin si**
6: **sinon**
7: **pour chaque** $\mathcal{P}_{w,k} = \langle X_{w,k}, \mathcal{D}_{w,k}, \mathcal{C}_{w,k} \rangle \in$ `Couper`$(\mathcal{P}_w)$ **faire**
8: **tant que** `Propagateurs` $\neq \emptyset$ **faire**
9: Soit $\mathfrak{P} \in$ `Propagateurs`
10: Exécuter $\mathfrak{P}$
11: **fin tant que**
12: `Résoudre`$(\mathcal{P}_{w,k})$
13: **fin pour**
14: **fin si**
15: **Fin fonction**

3.5 Optimisation

Les problèmes de programmation par contraintes ont en général plusieurs solutions, et il est souvent naturel dans considérer certaines comme meilleures que d'autres, ce qui conduit à la notion d'optimisation [35]. Pour cela, il est seulement nécessaire de définir une fonction objectif qui permet d'ordonner les solutions selon leur qualité.

Comme le nombre de solutions d'un CSP est fini, une façon de faire consiste à énumérer l'ensembles des solutions, et de ne garder que les meilleures. Si la fonction objectif peut être exprimée comme une contrainte, il devient possible de se souvenir de la valeur de la qualité de la meilleure solution trouvée, et de contraindre les solutions suivantes à être au moins meilleures que la dernière solution trouvée. Ceci permet d'effectuer naturellement de la propagation sur la fonction objectif.

La figure 3.3 décrit un algorithme générique pour l'optimisation sous contraintes. Dans cet algorithme, la fonction objectif est notée f, le propagateur associé à la contrainte objectif est notée $\mathfrak{F}$ et est paramétré par une valeur objectif, et la meilleure solution trouvée est notée $\mathcal{S}$.

Nous de détaillons pas outre mesure la problématique de l'optimisation dans le contexte des CSPs.

Algorithme 3.3 Optimisation sous contraintes

Fonction `Résoudre_r`($\mathcal{P}_w = \langle \mathcal{X}_w, \mathcal{D}_w, \mathcal{C}_w \rangle$: `CSP`)
  si $\forall x \in \mathcal{X}_w, |\mathcal{D}_w| \leqslant 1$ **alors**
    si $|\mathrm{Sol}(\mathcal{P}_w)| = 1$ **alors**
      $\mathcal{S} = \mathrm{Sol}(\mathcal{P}_w)$
    fin si
  sinon
    pour chaque $\mathcal{P}_{w,k} = \langle X_{w,k}, \mathcal{D}_{w,k}, \mathcal{C}_{w,k} \rangle \in$ `Couper`$(\mathcal{P}_w)$ **faire**
      tant que `Propagateurs` $\neq \emptyset$ **faire**
        Soit $\mathfrak{P} \in$ `Propagateurs` $\cup \{\mathfrak{F}(f(\mathcal{S}))\}$
        Exécuter $\mathfrak{P}$
      fin tant que
      `Résoudre_r`$(\mathcal{P}_{w,k})$
    fin pour
  fin si
Fin fonction
Fonction `Résoudre`($\mathcal{P}_\varepsilon = \langle \mathcal{X}_\varepsilon, \mathcal{D}_\varepsilon, \mathcal{C}_\varepsilon \rangle$: `CSP`)
  $\mathcal{S} = \bot$
  `Résoudre_r`$(\mathcal{P}_\varepsilon)$
  renvoyer $\mathcal{S}$
Fin fonction

3.6 Exemple

CSPLib [58] est une collection en ligne de problèmes de satisfaction de contraintes. L'un de ces problèmes est le problème d'ordonnancement d'une chaîne de production automobile [127], initialement présenté dans [99]. Il s'agit d'une simplification d'un problème dont de nombreuses variantes sont utilisées dans l'industrie [129]. Nous présentons cet exemple d'une part afin de décrire un problème de programmation par contraintes, et d'autre part parce que ce problème peut facilement être adapté au cas des varialbe flux. Cette nouvelle variante est décrite dans la section 8.2.

Le problème d'ordonnancement d'une chaîne de production automobile (OCPA) modélise une chaîne de montage automobile, équipée de stations capables d'aménager des voitures avec divers équipements. À titre d'exemple, les tâches peuvent consister à installer l'air conditionné, ou bien aménager un toit ouvrant. Un type de voiture est décrit par l'ensemble des équipements qui devront être installé pour ce type. Certains équipements sont plus faciles à installer que d'autres, donc certaines stations seront plus lentes que d'autres. Cependant, comme toutes les voitures ne demandent pas tous les équipements, la séquence des véhicules peut être agencée de telle sorte qu'un maximum de voitures puissent être traitées en un temps donné. Enfin, l'utilisateur souhaite traiter un nombre précis de voitures de chaque type, et il ne doit y avoir aucun délai entre le traitement d'une voiture et la suivante, en considérant un modèle temporel dans lequel chaque opération nécessite un nombre entier d'unités de temps.

station	1	2	3	4	5
capacité	1/2	2/3	1/3	2/5	1/5

TABLE 3.1 – Capacités des stations d'une instance du problème OCPA

		équipements				
type	quantité	1	2	3	4	5
1	1	✓		✓	✓	
2	1				✓	
3	2		✓			✓
4	2		✓		✓	
5	2	✓		✓		
6	2	✓	✓			

TABLE 3.2 – Types de voitures dans une instance du problème de OCPA est nombre de voitures de chaque types qui doivent être produites.

3.6.1 Formulation

La description précédente est générique, mais chaque instance est constituée d'un nombre différents d'équipements, pour lesquels les stations ont des capacités variées. La capacité d'une station est décrite par le nombre de voitures consécutives qu'elle peut traiter pour une sous-séquence de longueur spécifique. La formulation présentée ici correspond à l'instance décrite dans [40].

Dans cette instance, la chaîne de montage permet d'aménager cinq équipements, numérotés de 1 à 5. Les capacités des stations correspondantes sont données dans la table 3.1. Elles sont données sous la forme k/n, qui signifie que k voitures peuvent être traitées par cette station pour n voitures consécutives dans une séquence.

Les types de voitures sont décrits par les équipements qu'ils nécessitent, et sont accompagnés, dans la table 3.2, des nombres de voitures de chaque type.

Cette instance demande de trouver une séquence de dix voitures, dans laquelle, par exemple, exactement deux voitures sont de type numéro 3. Les voitures de type numéro 3 doivent passer par les stations numéros 2 et 5. La station numéro 5 ne peut traiter qu'une voiture toutes les cinq voitures, donc les deux voitures de type numéro 3 doivent être au moins séparées par quatre autres voitures. Si elles sont séparées d'exactement quatre voitures, aucune de ces voitures ne peut recevoir l'équipement fourni par la station numéro 5.

Cette instance admet une unique solution, donnée dans la table 3.3. Les descriptions complètes des types est données en plus de leurs désignations, afin de laisser la possibilité au lecteur de s'assurer que toutes les contraintes sont bien satisfaites. En particulier, il est aisé de constater que pas plus d'une voiture ne reçoit l'équipement fourni par la station numéro 5 dans une sous-séquence de cinq voitures.

3.6.2 Modélisation

Pour effectuer la modélisation d'une instance du problème OCPA en programmation par contraintes, il est nécessaire de définir les variables, leurs domaines, ainsi que les contraintes.

	Équipements				
Type	1	2	3	4	5
1	✓		✓	✓	
2				✓	
6	✓	✓			
3		✓			✓
5	✓		✓		
4		✓		✓	
4		✓		✓	
5	✓		✓		
3		✓			✓
6	✓	✓			

TABLE 3.3 – Une solution d'une instance du problème de d'ordonnancement de chaîne de production automobile.

Il est en général possible d'établir plusieurs modélisations équivalentes d'un même problème. Nous donnons ici une modélisation intuitive de l'instance que nous considérons ici, à titre d'exemple, mais il en existe de meilleures [63].

Cette modélisation est constitutée de cinquante variables booléenes $\mathtt{V}_{ij}$, agencées en une grille de 5×10, qui sont vraies si et seulement si la station numéro j doit traiter la i-ème voiture. Dix variables $\mathtt{T}_i \in \{1, \ldots, 6\}$ sont également utilisées afin d'établir les contraintes sur le nombre de voitures de chaque type dans la séquence.

La contrainte extensionnelle qui décrit les types de voitures désirés est utilisée dix fois le long de la séquence. Il s'agit d'une contrainte globale, qui impose que des variables ne prennent que des combinaisons de valeurs présentes dans une table donnée, ce qui pourrait être écrit de la façon suivante :

$$\texttt{extensionnelle}(\mathtt{T}, \langle \mathtt{T}_i, \mathtt{V}_{i1}, \mathtt{V}_{i2}, \mathtt{V}_{i3}, \mathtt{V}_{i4}, \mathtt{V}_{i5} \rangle)$$

où $\mathtt{T}$ est une table dont le contenu est équivalent à la table 3.2 privée de la deuxième colonne, et i prend les valeurs de 1 à 10.

Les contraintes sur les capacités des stations sont spécifiées par la contrainte de somme comparée, qui est aussi une contrainte globale. Elle impose qu'une somme de valeurs de variables soit plus grande ou plus petite qu'une certaine valeur. À l'écrit, il est préférable d'utiliser les notations mathématiques usuelles.

L'ensemble des contraintes de capacités sur les stations est décrit par la formule très concise suivante :

$$\sum_{k=0}^{n_j - 1} \mathtt{V}_{i+k,j} \leqslant k_j, \qquad \begin{array}{l} 1 \leqslant j \leqslant N_s \\ 1 \leqslant i \leqslant N - n_j \end{array}$$

où N est le nombre total de voitures, N_s est le nombre de stations, k_j est la capacité maximale de la station numéro j, et n_j est la longueur de la séquence sur laquelle cette capacité est prise pour la station numéro j. Chaque station engendre $N - n_j$ contraintes de capacité, donc le nombre total de contraintes est donné par :

$$\sum_{j=1}^{N_s} (N - n_j)$$

Enfin, il reste à spécifier le nombre de voiture de chaque type. Il existe des contraintes de dénombrement qui permettent d'imposer que le nombre de variables prenant une valeur spécifiée soit plus petit, égal, ou plus grand qu'une valeur donnée :

$$|\{i \in \{1, \ldots, N\} \mid \mathtt{T}_i = t\}| = n_t, \quad 1 \leqslant t \leqslant N_t$$

où N est le nombre de voitures, N_t est le nombre de types de voitures, et n_t est le nombre de voitures de types t souhaité.

3.7 Formalismes apparentés

La programmation par contraintes consiste à exprimer un problème comme une conjonction de contraintes sur des variables dont les domaines sont finis, avec une notion de contrainte générique. Il existe d'autre formalismes qui utilisent des techniques spécifiques à des problèmes de satisfaction de contraintes particuliers, parmi lesquels la programmation linéaire, la programmation entière et la programmation logique.

3.7.1 Programmation linéaire

La programmation linéaire est l'une des techniques de satisfaction de contraintes qui se distingue le plus de la programmation par contraintes sur les éléments finis. Dans ce formalisme, les domaines des variables sont les nombres flottants, qui sont des approximation des nombres réels, et les contraintes sont restreintes à des contraintes arithmétiques linéaires, associée à une fonction d'optimisation linéaire.

Les contraintes sont de la forme :

$$\sum c_i x_i \ \mathcal{R} \ c_r$$

avec c_i des coefficients constants, x_i des variables, c_r une constante, et $R \in \{\leqslant, =, \geqslant\}$ un symbole relationnel.

La fonction objectif est de la forme :

$$f = \sum c_i x_i$$

et l'objectif est de minimiser ou maximiser la valeur de cette fonction en respectant les contraintes.

La méthode de résolution de programmes linéaires la plus utilisée est l'algorithme du simplex [34, 150]. Contrairement au problème de satisfaction de contraintes sur les domaines finis, le problème de satisfaction linéaire existe depuis plusieurs décennies.

Le principe de la méthode du simplex consiste à démarrer sur une instanciation arbitraire, qui n'est pas optimale, puis à modifier les valeurs des variables selon une variation de l'algorithme de Gauss pour la résolution de système linéaires, afin d'aboutir à une solution optimale. La complexité du problème de satisfaction de contraintes linéaires sur les flottant est polynomiale en temps.

3.7.2 Programmation entière

La programmation entière (*integer programming*) est le pendant de la programmation linéaire pour les entiers. Le principe consiste à utiliser la programmation linéaire pour calculer une solution d'un problème linéaire sur les flottants, puis de modifier cette solution afin de calculer la solution optimale la plus proche sur les entiers. Le problème de satisfaction de contraintes linéaires sur les entiers est NP-complet, ce qui s'explique par le fait qu'étant donné une solution réelle satisfaisant les contraintes, il existe un nombre exponentiel d'approximations possibles sur les entiers.

Certains solveurs de contraintes, tels qu'ECLiPSe, sont conçus de telle sorte à pouvoir combiner les méthodes de résolution sur les domaines finis avec les algorithmes d'optimisation linéaires fournis par des librairies telles que COINOR [88], CPLEX [73] ou Xpress MP [51].

3.7.3 Programmation logique par contraintes

La programmation logique par contraintes [90, 49] s'appuie sur la programmation logique pour résoudre des problèmes de satisfaction de contraintes. Le problème de satisfaction de contraintes conjonctives est un cas particulier en programmation logique, car le langage est capable de gérer aussi la disjonction et la négation de contraintes. Cependandant, le support algorithmique pour ces deux opérateurs logiques est pour le moment loin d'être optimal ou satisfaisant dans un contexte général, bien qu'il soit utile pour des applications particulières.

Disjonction de contraintes Le problème de la disjonction de contraintes est que tant que les deux contraintes peuvent être vraies, il est impossible de propager quoi que ce soit. Le mécanisme de disjonction constructive [153] consiste à effectuer indépendamment la propagation de chaque contrainte. Si une valeur est un support dans aucune des contraintes après propagagation, alors elle peut être retirée. Bien que cette approche soit simple à mettre en œuvre, le coût d'exécution des propagateurs est en général trop élevé, ce qui rend le mécanisme inadéquat en pratique. Une méthode plus efficace [17] consiste à calculer un support pour chaque valeur de chaque variable, mais ceci impose d'être capable de calculer des supports spécifiques. Les propagateurs de contraintes dans Gecode sont spécifiés par des algorithmes dédiés qui effectuent seulement de la propagation, et ne proposent aucun moyen de répondre simplement à une telle question. De plus, pour des variables tels que les entiers, dont le domaine peut être vaste, et qui sont représentées par des intervalles, il serait préférable que le propagateur soit capable de déterminer des intervalles de supports valides afin de rendre l'approche viable.

Une autre approche pour gérer la disjonction consiste à utiliser des contraintes réifiées. Cette approche introduit un niveau d'indirection en reliant le fait qu'une contrainte est satisfaite à une variable booléenne b. Si la contrainte est satisfaite, alors b doit être positionnée à *vrai*. Si elle n'est pas satisfaite, alors b doit être positionnée à *faux*. Inversement, si b est positionnée à *vrai*, alors la contrainte doit être satisfaite, ce qui consiste à simplement propager la contrainte. En revanche, si b est positionnée à *faux*, la négation de la contrainte doit être propagée. Ce niveau d'indirection offre une flexibilité, car il permet d'utiliser des propagateurs dédiés pour les contraintes réifiées, et un propagateur générique pour propager la disjonction. La plupart des solveurs proposent des contraintes réifiées, mais limités seulement à certaines contraintes, et l'intérêt pour la réification de contraintes en général semble assez récent dans la littérature [16].

La disjonction est donc naturelle en programmation logique par contraintes, mais en général inefficace. Dans les solveurs du type de Gecode, la disjonction de contraintes est implémentées de façon partielles. Il est donc préférable pour le moment de considérer qu'un problème de satisfaction de contraintes sur les domaines finis est constitué d'une conjonction de contraintes, même si la réification de contraintes semble en voie de devenir généralisée dans un avenir proche.

Négation La négation de contraintes pose des problèmes subtils en programmation logique par contraintes [6], car la négation correspond à l'absence de solution, ce qui implique implicitement d'énumérer toutes les solutions possibles au lieu d'effectuer de la propagation. En ce qui concerne les solveurs basés sur des principes similaires à ceux de Gecode, la négation de contraintes nécessite la mise en œuvre de propagateurs dédiés. Il est à noter que la présence de contraintes réifiées implique naturellement la possibilité d'effectuer la négation de contraintes.

Le support algorithmique pour la négation de contraintes n'est donc pas aussi solide et efficace que pour la conjonction de contraintes, et il est donc préférable d'éviter d'utiliser la négation pour modéliser des problèmes de satisfaction de contraintes.

Il existe certaines contraintes pour lesquelles la négation est très simple à effectuer, telles que les contraintes arithmétiques basées sur les comparateurs '$<$', '$\leqslant$', '$\geqslant$', et '$>$'. Pour ces types de contraintes, il est tout aussi simple d'imposer à l'utilisateur d'utiliser le comparateur qui convient, plutôt que d'introduire le concept de négation.

3.8 Conclusion

Dans les chapitres suivants, nous considérerons que la programmation par contrainte consiste à modéliser des problèmes comme des conjonctions de contraintes sur des variables à domaines finis, dans lesquels les contraintes sont représentées par des propagateurs qu'il est seulement posssible d'exécuter. En particulier, nous considérerons que la disjonction et la négation ne font pas partie de la PPC, en regard de la situation actuelle concernant la gestion de ces deux connecteurs logiques dans la plupart des solveurs de contraintes. Ceci contraste beaucoup avec la forte connotation logique qui sera présente tout au long de ce manuscrit, car la négation et la disjonction y sont généralement considérés comme des opérateurs de base qu'il est inhabituel de remettre en question.

Le chapitre suivant concerne les logiques classiques et les logiques temporelles. Ces dernières sont très faiblement discutées dans la littérature sur la programmation par contraintes, ce qui peut paraître surprenant, car les deux formalismes ont tout deux pour objectif de décrire des ensembles d'objets satisfaisants des propriétés. La difficulté principale réside dans le fait que les objets manipulés dans le second cas sont de tailles infinies. Une manière de concicilier ces deux mondes sera décrite dans les contributions.

Chapitre 4

Logiques temporelles

Sommaire

4.1 Introduction

Les logiques temporelles sont très utilisées pour spécifier des ensembles de séquences infinies. Ces logiques sont principalement utilisées afin de résoudre le problème du *model checking* [9], qui consiste à vérifier qu'une implémentation logicielle ou matérielle satisfait une spécification de son comportement. En revanche, elles sont très peu étudiées dans le contexte de la programmation par contraintes, et cette difficulté est augmentée par l'existence d'un grand nombre de formalismes temporels aux caractéristiques différentes.

Nous présentons dans ce chapitre une vision unifiée de la logique classique et des logiques temporelles. Nous présentons d'abord un rappel des notions de logique classique dans la section 4.2, puis nous présentons les logiques temporelles dans la section 4.3, où les aspects importants concernant la logique temporelle linéaire sont développés selon une perspective historique. Nous détaillons ensuite une méthode algorithmique pour la résolution du problème de satisfaction de la logique temporelle linéaire dans la section 4.4, puis nous terminons par un bilan des logiques temporelles qui sont apparentées à celle-ci dans la section 4.5. Cette synthèse motive le choix qui est fait dans les contributions de se focaliser principalement sur la logique temporelle linéaire pour l'étude du problème de satisfaction de contraintes sur les variables flux.

4.2 Logiques classiques

La logique classique comprend la logique propositionnelle, la logique propositionnelle quantifiée ainsi que les logiques du premier ordre, du second ordre et d'ordres supérieurs. Nous ne traitons pas ici des logiques d'ordres supérieurs à 2, car les travaux auxquels nous ferons référence ne les utilisent pas. Une présentation de celles-ci figure dans [85].

Pour chaque logique ou famille de logiques, nous présentons leur syntaxe, leur sémantique, ainsi que quelques observations concernant les problèmes de vérification et de satisfaction pour ces logiques.

4.2.1 Logique propositionnelle

Les formules de la logique propositionnelle sont formées de combinaisons de variables propositionnelles avec des connecteurs logiques. Le concept de variable propositionnelle est à la base de toutes les logiques classiques, car elle constitue l'abstraction la plus générale de la notion de valeur de vérité booléenne. Les logiques d'ordres supérieures induisent une structure plus élaborée sur le concept de variable propositionnelle.

Syntaxe de la logique propositionnelle. Soit $\mathcal{P}$ un ensemble dénombrable de variables propositionnelles à valeurs dans $\mathbb{B}$. Les règles de formation des formules de la logique propositionnelle sont les suivantes :

— Une variable propositionnelle $x \in \mathcal{P}$ est une formule ;

— Si φ et θ sont deux formules, alors $\varphi \barwedge \theta$ est une formule.

Nous notons $\mathrm{var}(\varphi) \subset \mathcal{P}$ le sous-ensemble des variables de $\mathcal{P}$ qui apparaissent dans une formule de logique propositionnelle φ.

La notion d'interprétation en logique est similaire à la notion d'instanciation utilisée en programmation par contraintes (Définition 18).

$\neg\varphi$	Négation	$\neg\varphi \equiv \varphi \barwedge \varphi$
$\varphi \wedge \theta$	Conjonction	$\varphi \wedge \theta \equiv \neg(x \barwedge y)$
$\varphi \vee \theta$	Disjonction	$\varphi \vee \theta \equiv \neg((\neg\varphi) \wedge (\neg\theta))$
$\varphi \leftrightarrow \theta$	Equivalence	$\varphi \leftrightarrow \theta \equiv (\varphi \rightarrow \theta) \wedge (\theta \rightarrow \varphi)$
$\varphi \oplus \theta$	Disjonction exlusive	$\varphi \oplus \theta \equiv (\varphi \vee \theta) \wedge (\neg(\varphi \wedge \theta))$
$\varphi \barvee \theta$	Disjonction inversée	$\varphi \barvee \theta \equiv \neg(\varphi \vee \theta)$
$\varphi \rightarrow \theta$	Implication	$\varphi \rightarrow \theta \equiv (\neg\varphi) \vee \theta$
$\varphi \leftarrow \theta$	Explication	$\varphi \leftarrow \theta \equiv \varphi \vee (\neg\theta)$
$\varphi \nrightarrow \theta$	Implication inversée	$\varphi \nrightarrow \theta \equiv \neg(\varphi \rightarrow \theta)$
$\varphi \nleftarrow \theta$	Explication inversée	$\varphi \nleftarrow \theta \equiv \neg(\varphi \leftarrow \theta)$
$\top$	Vérité	$\top \equiv \varphi \vee \neg\varphi$ φ quelconque
$\bot$	Contrevérité	$\bot \equiv \varphi \wedge \neg\varphi$ φ quelconque

FIGURE 4.1 – Connecteurs propositionnels unaires et binaires définis à partir de la conjonction inversée ($\barwedge$)

Définition 35 (Interprétation propositionnelle)**.** Soit φ une formule propositionnelle. Une **interprétation** des variables de φ est une fonction $I : \text{var}(\varphi) \rightarrow \mathbb{B}$ qui associe à chaque variable de φ une valeur de vérité. L'ensemble des interprétations de $\text{var}(\varphi)$ est noté $\mathcal{I}(\text{var}(\varphi))$. Le **domaine** de I est noté $\text{dom}(I)$. La **projection** de I sur $Y \subseteq \text{dom}(I)$ est l'unique instanciation $I{\downarrow}Y$ telle que $(I{\downarrow}Y)(y) = I(y)$ pour $y \in Y$ et $\text{dom}(I{\downarrow}Y) = Y$.

Sémantique de la logique propositionnelle. La sémantique de la logique propositionnelle est la suivante :

— $I \vDash x$ si et seulement si $I(x) = \top$ pour $x \in \mathcal{P}$ et $I \in \mathcal{I}(\{x\})$.

— $I \vDash \varphi \barwedge \psi$ si et seulement si $(I{\downarrow}\text{var}(\varphi)) \nvDash \varphi$ ou $(I{\downarrow}\text{var}(\psi)) \nvDash \psi$ pour $I \in \mathcal{I}(\text{var}(\varphi) \cup \text{var}(\psi))$

Définition 36 (Modèle)**.** Soit φ une formule de logique propositionnelle, $X \subset \mathcal{P}$ tel que $\text{var}(\varphi) \subseteq X$, et $I \in \mathcal{I}(X)$ une interprétation des variables de X. I est un **modèle** de φ si et seulement si $(I \downarrow \text{var}(\varphi)) \vDash \varphi$. L'ensemble des **modèles exacts** de φ est noté $\text{Mod}(\varphi)$.

$$\text{Mod}(\varphi) = \{I \in \mathcal{I}(\text{var}(\varphi)) \mid I \vDash \varphi\}$$

Un connecteur propositionnel unaire est une fonction de type $\mathbb{B} \rightarrow \mathbb{B}$, et un connecteur propositionnel binaire est une fonction de type $\mathbb{B}^2 \rightarrow \mathbb{B}$. Étant données deux formules de logique propositionnelle φ et θ, il est possible de définir deux connecteurs logiques unaires et seize connecteurs logiques binaires. Nous donnons les définitions de ces connecteurs par des équivalences basées sur la conjonction inversée ($\barwedge$) dans la figure 4.1, ainsi que les tables de vérité de ces connecteurs dans la figure 4.2. Les connecteurs logiques les plus utilisés en pratique sont la négation ($\neg$), la conjonction ($\wedge$), la disjonction ($\vee$), l'implication ($\rightarrow$) et l'équivalence ($\leftrightarrow$). Les noms et symboles des autres connecteurs ne sont pas standardisés, malgré l'usage assez répandu des connecteurs '$\barwedge$' et '$\oplus$'.

Un ensemble de connecteurs logiques forme une base s'il est possible de définir tous les autres connecteurs à partir de ceux de la base. La base $\{\barwedge\}$ est très utilisée en électronique [82] car elle permet de définir la logique propositionnelle avec une unique porte logique. La base $\{\barvee\}$ est une autre base possible. En algorithmique, il est plus fréquent d'utiliser la base $\{\wedge, \vee, \neg\}$.

φ	θ	$\bot$	$\varphi \wedge \theta$	$\varphi \nrightarrow \theta$	φ	$\varphi \nleftarrow \theta$	θ	$\varphi \oplus \theta$	$\varphi \vee \theta$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$

φ	θ	$\varphi \barwedge \theta$	$\varphi \leftrightarrow \theta$	$\neg\theta$	$\varphi \leftarrow \theta$	$\neg\varphi$	$\varphi \rightarrow \theta$	$\varphi \barvee \theta$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\top$

FIGURE 4.2 – Tables de vérité des opérateurs unaires et binaires en logique propositionnelle

En particulier, toute formule de logique propositionnelle peut être présentée sous forme normale conjonctive.

Définition 37 (Littéral)**.** Soit $\mathcal{P}$ un ensemble dénombrable de variables propositionnelles et $X \subseteq \mathcal{P}$. Un **littéral** est une variable $x \in \mathcal{P}$ ou sa négation $\neg x$. L'**ensemble des littéraux** associé à X est noté

$$L(X) = \bigcup_{x \in X} \{x, \neg x\}$$

Un littéral $l \in L(\mathcal{P})$ est **positif** si $l \in \mathcal{P}$, et est **négatif** sinon. Nous notons $L^+(X) = L(X) \cap \mathcal{P}$ et $L^-(X) = L(X) \backslash \mathcal{P}$.

Définition 38 (Forme normale conjonctive)**.** Une formule est sous **forme normale conjonctive** si elle a la structure d'une conjonction de clauses. Une **clause** est une disjonction de littéraux.

Les clauses de Horn sont des clauses particulières pour lesquelles de nombreux problèmes de logique propositionnelle sont simplifiés. Ces clauses sont à la base de la programmation logique.

Définition 39 (Clause de Horn)**.** Soit $\mathcal{P}$ un ensemble dénombrable de variables propositionnelles. Une **clause de Horn** est une disjonction de littéraux contenant au plus un littéral positif. L'ensemble des clauses de Horn sur $\mathcal{P}$ est définissable de la façon suivante :

$$\mathrm{Horn}(\mathcal{P}) = \left\{ \bigvee_{1 \leqslant k \leqslant |w|} w[k] \mid w \in L^-(\mathcal{P})^* \cdot \mathcal{P} \cdot L^-(\mathcal{P})^* \right\}$$

Il est souvent nécessaire de considérer des formules logiques dans lesquelles certaines variables ont été substituées par d'autres formules. Nous donnons ici la définition d'une substitution pour les variables libres d'une formule de logique propositionnelle. Cette définition doit être adaptée dans le contexte d'autres logiques, notamment celles dont les formules comportent des variables liées à des quantificateurs.

Définition 40 (Substitution)**.** Soit φ une formule de logique propositionnelle et $x \in \mathrm{var}(\varphi)$. La formule $\varphi\left[\frac{x}{\theta}\right]$ représente la formule dans laquelle la formule θ a été substituée à la variable x.

Exemple 4. Soit $\varphi = (x \vee y) \wedge (y \rightarrow z)$. Alors

$$\varphi\left[\frac{y}{u \vee v}\right] = (x \vee (u \vee v)) \wedge ((u \vee v) \rightarrow z)$$

4.2.2 Problème de vérification et problème de satisfaction

Pour une logique, le problème de satisfaction consiste à déterminer si une formule φ donnée admet un modèle, et le problème de vérification consiste à déterminer, étant données une formule φ et une instanciation $I \in \mathcal{I}(\text{var}(\varphi))$, si I est un modèle de φ.

Pour la logique propositionnelle, le problème de satisfaction est extrêmement simple, puisqu'il peut être résolu en temps linéaire par simple subsitution des variables puis évaluation. Ceci n'est pas vrai pour des logiques plus élaborées.

Contrairement au problème de vérification, le problème de satisfaction de la logique propositionnelle (SAT) est au cœur de la théorie de la complexité, car il s'agit du problème caractéristique de la classe des problèmes NP-complets [32]. Les solveurs SAT sont généralement construits pour résoudre des formules de logique propositionnelle mises sous forme normale conjonctive. Cette représentation rend le problème SAT pour la logique propositionnelle très proche du problème de satisfaction de contraintes, car un CSP est une conjonction de contraintes, dans lesquelles les disjonctions sont dissimulées dans les algorithmes de propagation. Tout comme en programmation par contraintes, la réalisation de solveurs efficaces pour la résolution du problème SAT est un sujet de recherche toujours d'actualité [154].

4.2.3 Problème de synthèse

Le problème de synthèse pour une formule propositionnelle φ peut être résolu avec un nombre de résolutions du problème de satisfaction linéaire en le nombre de variables. En effet, soit $x \in \text{var}(\varphi)$ une variable de φ. Si φ est satisfaisable, alors soit $\varphi\left[\frac{x}{\top}\right]$ est satisfaisable, soit $\varphi\left[\frac{x}{\bot}\right]$ est satisfaisable. Il suffit donc d'effectuer la première substitution qui convient, puis continuer jusqu'à ce que toutes les variables soient instanciées, pour obtenir un modèle de φ.

En ce qui concerne la programmation par contraintes, les même considérations s'appliquent, car le problème de satisfaction de contraintes est lui aussi NP-complet. Du point de vue de la théorie de la complexité, les problèmes de satisfaction et de synthèse sont donc équivalents pour la logique propositionnelle et la programmation par contraintes, mais les solveurs de contraintes et les solveurs SAT résolvent en réalité le problème de synthèse, car l'objectif est de produire un modèle explicite de la formule ou du problème. L'algorithme de recherche en profondeur (Algorithme 3.1) est une implémentation exacte du procédé décrit dans le précédent paragraphe.

Il n'est pas évident que ce principe s'applique à toutes les logiques. En particulier, il existe des logiques pour lesquelles le problème de satisfaction est décidable, alors que leurs modèles sont de tailles infinies, et c'est précisément le cas pour les logiques temporelles qui font l'objet des sections suivantes. Nous nous abstiendrons pour le moment d'énoncer quelques propriétés générales que ce soit concernant le lien entre la complexité du problème de satisfaction pour une logique et la complexité de son problème de synthèse.

4.2.4 Logiques du premier ordre

Les logiques du premier ordre sont caractérisées par l'utilisation des quantificateurs universels et existentiels. L'introduction de la quantification impose un domaine $\mathcal{D}$ parmi lequel choisir des valeurs pour les variables, ce qui, mis à part le cas particulier de la logique propositionnelle quantifiée, impose de distinguer les valeurs des variables des valeurs de vérités des formules. Le

lien entre ces deux types de valeurs se fait par l'intermédiaire de prédicats, qui sont des fonctions de domaine $\mathcal{D}^k$ à valeurs dans $\mathbb{B}$.

Un premier critère de classification des logiques du premier ordre est la taille du domaine de quantification. Celui-ci peut-être fini ou infini. S'il est fini, il peut contenir deux, ou plusieurs valeurs. Un domaine à moins de deux valeurs rend la logique inintéressante, car les choix de valeurs disponibles pour la quantification deviennent trop limités.

Un domaine à deux valeurs est en bijection avec le domaine des valeurs de vérités, et il est alors naturel de les considérer identiques, ce qui rend possible d'imbriquer les formules dans les prédicats. Le nombre de prédicats possibles est en revanche très limité, si bien que la notion de prédicat n'est plus nécessaire dans ce contexte, et peut être remplacée par la notion de connecteur logique. Ce cas particulier correspond à la logique proposionnelle quantifiée, qui sera traité à part.

Le cas plus général des domaines finis avec au moins trois valeurs nécessite la notion de prédicats. Cependant, pour ces domaines, il est possible d'effectuer des transformation vers la logique propositionnelle quantifiée, du même type que la transformation effectuée pour modéliser le problème de satisfaction de contraintes par une formule de logique propositionnelle, ce qui rend les logiques du premier ordre sur les domaines finis très similaires à la logique propositionnelle quantifiée. La programmation logique correspond à la généralisation des clauses de Horn à la logique du premier ordre sur les domaines finis. La sémantique de "monde clos" usuellement évoquée pour justifier les procédure décisions correspond à l'hypothèse d'un domaine fini, ce qui rend possible d'examiner les valeurs de vérités de tous les termes du premier ordre.

Concernant les domaines de tailles infinies, ceux-ci peuvent être dénombrables s'il s'agit des entiers naturels, des entier relatifs, ou des nombres rationnel, ou indénombrables s'il s'agit des nombres réels ou complexes. Le cas le plus étudié est celui des entiers naturels, et la logique du premier ordre sur les entiers naturels est indécidable [65]. Ceci oblige à considérer des logiques du premier ordre portant sur des prédicats particuliers, dont la sémantique est donnée à l'avance, ou bien dont certaines propriétés sont données à l'avance. Il en va de même pour les logiques du premier ordre qui portent sur d'autres domaines de tailles infinies.

Définition 41 (Prédicat)**.** Un **prédicat** sur un ensemble $\mathcal{D}$ est une fonction $\mathcal{P} : \mathcal{D}^k \to \mathbb{B}$. Le nombre de valeurs k sur lesquelles porte $\mathcal{P}$ est appelé l'**arité** de $\mathcal{P}$. Il est usuel de noter un prédicat $\mathcal{P}$ d'arité k sous la forme $\mathcal{P}/k$. Pour un prédicat P, nous notons son arité $\mathrm{ar}(P)$.

Définition 42 (Structure)**.** Une **structure** est une paire $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ dans laquelle $\mathcal{D}$ est un ensemble appelé **domaine** de $\mathcal{S}$, et $\mathcal{P} = \{P_i/k_i\}_{1 \leqslant i \leqslant n}$ est un ensemble fini de n prédicats.

Définition 43 (Terme)**.** Soit $\mathcal{V}$ un ensemble dénombrable de variables. Un **terme** $\psi = P(v_1, \ldots, v_k)$ sur une structure $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ est l'application d'un prédicat $P \in \mathcal{P}$ d'arité $k = \mathrm{ar}(P)$ à un tuple de k variables $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$ à valeurs dans $\mathcal{D}$. L'ensembles des variables de ψ est $\mathrm{var}(\psi) = \{v_1, \ldots, v_k\}$.

Syntaxe de la logique du premier ordre. Soient $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ une structure et $\mathcal{V}$ un ensemble dénombrable de variables à valeurs dans $\mathcal{D}$. La **syntaxe** de la logique du premier ordre sur $\mathcal{S}$ est définie inductivement de la façon suivante :

— Un terme sur $\mathcal{S}$ est une formule ;

— Si ψ et θ sont des formules, alors $\psi \barwedge \theta$ est une formule ;

— Si φ est une formule, alors $\exists x.\varphi$ et $\forall x.\varphi$ sont des formules.

Le type de logique du premier ordre présenté ici est minimaliste. En particulier, il n'est pas fait mention de fonctions $\mathcal{D}^k \to \mathcal{D}$, car celles-ci peuvent être modélisées par des prédicats [43]. Le domaine de quantification étant unique, il est laissé implicite dans l'utilisation des quantificateurs et les connecteurs propositionnels autre que le connecteur '$\bar{\wedge}$' peuvent être intégrés en utilisant les équivalences de la figure 4.1. La définition habituelle distingue la notion de prédicat de celle de constante. Comme une constante est une fonction d'arité nulle à valeur dans $\mathcal{D}$, elle peut être modélisée par un prédicat unaire. La généralisation à des systèmes typés est assez simple mais nécessite de développer un système de notations plus élaboré [118].

L'introduction des quantificateurs nécessite de préciser la définition de l'ensemble des variables d'une formule. Les présentations habituelles distinguent la notion de variable libre de celle de variable liée. Une telle distinction implique implicitement de pouvoir identifer une variable liée dans une formule. Comme la portée d'une variable liée est définie par le quantificateur qui lui est associé, il est techniquement possible d'utiliser une même variable au lieu de plusieurs. Par exemple, la formule $\forall x, (P(x) \wedge \exists x \neg P(x))$ est correcte et équivalente à la formule $\forall x, (P(x) \wedge \exists y, \neg P(y))$. Ceci conduit certaines présentations à imposer que toutes les variables quantifiées soient différentes pour tous les quantificateurs, mais cette contrainte impose à son tour de définir une procédure pour effectuer l'α-conversion des formules lorsqu'elles sont utilisées dans d'autres formules. A l'opposé, certains travaux étudient l'expressivité de logiques restreintes à l'utilisation d'un nombre fixé de variables [62], ce qui impose de pouvoir réutiliser une même variable pour effectuer plusieurs quantifications. Dans la définition qui suit, nous donnons seulement la définition des variables libres, que nous identifions à la notion de variables d'une formule. En l'état, cette présentation possède l'inconvénient que la substitution d'une variable d'une formule par une formule peut masquer certaines variables liées par conflit avec les variables quantifiée. Par exemple, si $\varphi = P(x) \vee \forall y, (P(y) \to P(x))$, alors $\varphi\left[\frac{x}{y}\right] = P(y) \vee \forall y, (P(y) \to P(y))$ est une tautologie. Nous considérons néanmoins que le formalisme de la logique du premier ordre présenté ici a l'avantage de la simplicité, et nous considérons les conséquences décrites dans ce paragraphe comme des cas limites que nous éviterons dans la suite.

Définition 44 (Variables d'une formule du premier ordre)**.** Soit φ une formule du premier ordre sur une structure $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$. L'ensemble des **variables** de φ est défini récursivement comme suit :

- $\mathrm{var}(P(v_1, \ldots, v_k)) = \{v_1, \ldots, v_k\}$ pour $P \in \mathcal{P}$, $k = \mathrm{ar}(P)$ et $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$
- $\mathrm{var}(\psi \bar{\wedge} \theta) = \mathrm{var}(\psi) \cup \mathrm{var}(\theta)$;
- $\mathrm{var}(\exists x.\psi) = \mathrm{var}(\forall x.\psi) = \mathrm{var}(\psi) \backslash \{x\}$

Définition 45 (Interprétation du premier ordre)**.** Soit φ une formule du premier ordre sur une structure de domaine $\mathcal{D}$. Une **interprétation** de φ est une fonction $I : \mathrm{var}(\varphi) \to \mathcal{D}$. Le **domaine** de I et noté $\mathrm{dom}(I)$. L'ensemble des interprétations de φ est noté $\mathcal{I}(\varphi)$.

Définition 46 (Projection et extension d'une interprétation)**.** Soient $\mathcal{D}$ le domaine d'une structure, $\mathcal{V}$ un ensemble dénombrable de variables à valeurs dans $\mathcal{D}$, et I une interprétation. La **projection** de I sur $Y \subset \mathrm{dom}(I)$ est l'unique interprétation $I{\downarrow}Y$ telle que $(I{\downarrow}Y)(y) = I(y)$ pour tout $y \in Y$ et $\mathrm{dom}(I{\downarrow}Y) = Y$. Une **extension** de I à $Y \subset \mathcal{V} \cap \mathrm{dom}(I)$ est une interprétation I' de domaine $\mathrm{dom}(I') = \mathrm{dom}(I) \cup Y$ telle que $I'(y) = I(y)$ pour tout $y \in \mathrm{dom}(I)$. L'ensemble des extensions de I sur Y est noté $I{\uparrow}Y$.

Sémantique de la logique du premier ordre. Soit $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ une structure. La sémantique de la logique du premier ordre sur $\mathcal{S}$ est la suivante :

- $I \vDash P(v_1, \ldots, v_k)$ si et seulement si $P(I(v_1), \ldots, I(v_k)) = \top$, où $P \in \mathcal{P}$, $k = \text{ar}(P)$, $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$ et $I \in \mathcal{I}(\{v_1, \ldots, v_k\})$;
- $I \vDash \psi \barwedge \theta$ si et seulement si $(I{\downarrow}\text{var}(\psi)) \nvDash \psi$ ou $(I{\downarrow}\text{var}(\theta)) \nvDash \theta$, où $I \in \mathcal{I}(\text{var}(\psi \barwedge \theta))$
- $I \vDash \exists x.\psi$ si et seulement s'il existe une extension $I' \in (I{\uparrow}\{x\})$ de I telle que $I' \vDash \psi$, où $I \in \mathcal{I}(\text{var}(\exists x.\psi))$;
- $I \vDash \forall x.\psi$ si et seulement si pour toute extension $I' \in (I{\uparrow}\{x\})$ de I, $I' \vDash \psi$, où $I \in \mathcal{I}(\text{var}(\forall x.\psi))$

4.2.5 Logique propositionnelle quantifiée

La logique propositionnelle quantifiée est la logique du premier ordre sur les booléens $\mathbb{B}$. La notion de prédicat étant redondante, la logique propositionnelle quantifiée admet une définition de sa syntaxe plus élémentaire.

Syntaxe de la logique propositionnelle quantifiée. Soit $\mathcal{P}$ un ensemble dénombrable de variables à valeurs dans $\mathbb{B}$. La **syntaxe** de la logique du premier ordre sur $\mathcal{S}$ est la suivante :

- p est une formule, où $p \in \mathcal{P}$;
- Si ψ et θ sont des formules, alors $\psi \barwedge \theta$ sont des formules ;
- Si φ est une formule, alors $\exists x.\varphi$ et $\forall x.\varphi$ sont des formules.

Le problème de satisfaction de la logique propositionnelle quantifiée est le problème représentatif de la classe des problèmes PSPACE-complet [55]. Du fait de la quantification, le problème de vérification est identique au problème de satisfaction, car les variables quantifiées doivent tout de même être instanciées pour vérifier qu'une interprétation est un modèle. Concernant la problème de synthèse, une variante consiste à intégrer l'arbre de recherche au résultat, afin de produire un témoin, de taille exponentielle, que la solution renvoyée est bien un modèle. Ce témoin correspond à une stratégie dans le contexte des jeux, où le joueur universel correspond à l'adversaire.

Les QCSPs [29] sont des problèmes de satisfaction de contraintes quantifiées dont la syntaxe est celle de la logique du première ordre sur des domaines finis, et dans lesquels les prédicats sont des contraintes.

4.2.6 Logique du second ordre

Les logiques du second ordre sont les extensions des logiques du premier ordre par des quantificateurs du second ordre. Pour une structure de domaine $\mathcal{D}$, un quantificateur du premier ordre quantifie sur $\mathcal{D}$, et un **quantificateur du second ordre** d'arité k quantifie sur des relations d'arités k, c'est-à-dire sur $\text{Part}(\mathcal{D}^k)$.

Une logique du second ordre **monadique** est une logique du second ordre dans laquelle les domaines de quantifications des quantificateurs du second ordre sont restreints à des relations unaires, c'est-à-dire à des sous-ensemble de $\mathcal{D}$, et une logique du second ordre monadique **faible** est une logique du second ordre monadique pour laquelle les domaines de quantification sont restreints à des sous-ensembles finis, même si le domaine lui-même est de taille infinie.

Définition 47 (Ensembles de variables)**.** Afin d'alléger le texte, nous définissons les ensembles de variables suivants. $\mathcal{V}$ est un ensemble dénombrable de variables du premier ordre, et pour tout $k \geqslant 1$, $\mathcal{W}_k$ est un ensemble de variables du second ordre d'arités k, c'est-à-dire tel que pour tout $W \in \mathcal{W}_k$, $\mathrm{ar}(W) = k$. Nous notons $\mathcal{W} = \bigcup_{k \geqslant 1} \mathcal{W}_k$ l'ensemble des variables du second ordre.

Syntaxe de la logique du second ordre. Soit $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ une structure, et $\mathcal{V}$ et $\mathcal{W}$ définis selon la Définition 47. La syntaxe de la logique du second ordre est identique à celle de la logique du premier ordre, mis à part l'ajout des variables et des quantificateurs du second ordre :

— $P(v_1, \ldots, v_k)$ est un **terme**, où $P \in \mathcal{P}$, $k = \mathrm{ar}(P)$ et $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$;

— $W(v_1, \ldots, v_k)$ est un **terme**, où $W \in \mathcal{W}$, $k = \mathrm{ar}(W)$ et $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$;

— Un terme est une formule ;

— Si ψ et θ sont des formules, alors $\psi \barwedge \theta$ est une formule ;

— Si φ est une formule, alors $\exists \Upsilon.\varphi$ et $\forall \Upsilon.\varphi$ sont des formules, où $\Upsilon \in \mathcal{V} \cup \mathcal{W}$.

Les notions de variables et d'interprétations doivent être adaptées pour couvrir les variables du second ordre.

Définition 48 (Variables d'une formule du second ordre)**.** Soient $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ une structure, et φ une formule du second ordre sur $\mathcal{V}$ et $\mathcal{W}$. L'ensemble des **variables** de φ est défini comme suit :

— $\mathrm{var}(P(v_1, \ldots, v_k)) = \{v_1, \ldots, v_k\}$ pour $P \in \mathcal{P}$, $k = \mathrm{ar}(P)$, et $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$;

— $\mathrm{var}(W(v_1, \ldots, v_k)) = \{v_1, \ldots, v_k, W\}$ pour $W \in \mathcal{W}$, $k = \mathrm{ar}(W)$, et $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$;

— $\mathrm{var}(\psi \barwedge \theta) = \mathrm{var}(\psi) \cup \mathrm{var}(\theta)$;

— $\mathrm{var}(\exists \Upsilon, \psi) = \mathrm{var}(\forall \Upsilon, \psi) = \mathrm{var}(\psi) \backslash \{\Upsilon\}$ pour $\Upsilon \in \mathcal{V} \cup \mathcal{W}$.

La notion d'interprétation est un peu plus complexe pour la logique du second ordre.

Définition 49 (Interprétation en logique du second ordre)**.** Soient $\mathcal{D}$ un domaine de valeurs, et $X \subset \mathcal{V} \cup \mathcal{W}$ un ensemble de variables du premier et du second ordre. Une **inteprétation** des variables de X dans $\mathcal{D}$ est une fonction $I : X \to \mathcal{D} \cup \mathrm{Part}(\mathcal{D}^+)$ qui associe à chaque variable du premier ordre $v \in X \cap \mathcal{V}$ une valeur $I(v) \in \mathcal{D}$, et à chaque variable du second ordre $W \in X \cap \mathcal{W}$ un sous-ensemble $I(W) \in \mathrm{Part}(\mathcal{D}^k)$, où $k = \mathrm{ar}(W)$. Les notions de projection et d'extension définies pour la logique du premier ordre sont adaptées de façon similaire.

Sémantique de la logique du second ordre. Soit $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ une structure. La sémantique des formules du second ordre sur $\mathcal{S}$ est définie de la façon suivante :

— $I \vDash P(v_1, \ldots, v_k)$ si et seulement si $P(I(v_1), \ldots, I(v_k)) = \top$,
où $P \in \mathcal{P}$, $k = \mathrm{ar}(P)$, $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$ et $I \in \mathcal{I}(\{v_1, \ldots, v_k\})$;

— $I \vDash W(v_1, \ldots, v_k)$ si et seulement si $\langle I(v_1), \ldots, I(v_k) \rangle \in I(W)$,
où $W \in \mathcal{W}$, $k = \mathrm{ar}(W)$, $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$, $W \in \mathrm{Part}(\mathcal{V}^k)$ et $I \in \mathcal{I}(\{v_1, \ldots, v_k, W\})$;

— $I \vDash \psi \barwedge \theta$ si et seulement si $(I{\downarrow}\mathrm{var}(\psi)) \nvDash \psi$ ou $(I{\downarrow}\mathrm{var}(\theta)) \nvDash \theta$,
où $I \in \mathcal{I}(\mathrm{var}(\psi \barwedge \theta))$;

— $I \vDash \exists \Upsilon.\psi$ si et seulement s'il existe une extension $I' \in (I{\uparrow}\{\Upsilon\})$ de I telle que $I' \vDash \psi$,
où $\Upsilon \in \mathcal{V} \cup \mathcal{W}$ et $I \in \mathcal{I}(\mathrm{var}(\exists \Upsilon.\psi))$;

— $I \vDash \forall\Upsilon.\psi$ si et seulement si pour toute extension $I' \in (I\uparrow\{\Upsilon\})$ de I, $I' \vDash \psi$, où $\Upsilon \in \mathcal{V} \cup \mathcal{W}$ et $I \in \mathcal{I}(\text{var}(\forall\Upsilon.\psi))$.

Les logiques du second sont extrêmement générales, et pour la plupart indécidables. Cependant, les algorithmes de la théorie des logiques temporelles et de la théorie des langages sur les mots et arbres infinis est très liées à des logiques du second ordre monadiques particulières.

4.3 Logiques temporelles

L'utilisation actuelle des logiques temporelles fait appel à de nombreuses notions, qu'il est aussi simple d'introduire sous la forme d'une chronologie. La présentation que nous donnons ici est une version condensée de la chronologie [139]. De nombreux éléments figurent également dans [115].

Les logiques temporelles ont longtemps été considérées comme un domaine distinct de la logique classique. Les concepts utilisés dans les logiques temporelles actuelles sont en particulier déjà présents dans [108] (1952), mais le lien avec la logique classique ne sera établi que dans la thèse [75] (1968), où il y est en particulier fait état que la logique temporelle linéaire est équivalente à la logique du premier ordre sur la structure $\langle \mathbb{N}, \{=, <\}\rangle$, avec une transformation linéaire de la première vers la seconde, mais une transformation exponentielle de la seconde vers la première.

Parallèlement, le lien entre logique sur les entiers naturels et automates a été établi dans [24] (1960) pour les mots finis, puis étendu aux mots infinis dans [26] (1962), ainsi que dans un foisonnement de publications contemporaines de celles-ci, dont les références se trouvent dans [139]. L'équivalence entre logique sur les entiers naturels et automates nécessite l'introduction de la notion de structure de mot.

Définition 50 (Structure de mot [134])**.** Soit Σ un alphabet fini et $w \in \Sigma^*$ un mot sur Σ. La **structure de mot fini** associée à w est la structure $\mathcal{S}_w = \langle \mathcal{D}_w, \{=_w, <_w\} \cup \mathcal{P}_w\rangle$ dans laquelle :

— Le domaine $\mathcal{D}_w = [1 \dots |w|] = \{1, 2, \dots, |w|\}$ est l'intervalle de $\mathbb{N}$ qui permet d'indexer les lettres de w ;

— $=_w$ est la relation d'égalité sur $\mathcal{D}_w$, avec la même sémantique que pour les entiers naturels, mais restreinte à $\mathcal{D}_w$;

— $<_w$ est la relation d'ordre sur $\mathcal{D}_w$, avec la même sémantique que pour les entiers naturels, mais restreinte à $\mathcal{D}_w$;

— $\mathcal{P}_w$ est un ensemble de prédicats unaires $\mathcal{P}_w = \{P_s \mid s \in \Sigma\}$ tels que, pour chaque lettre $s \in \Sigma$, et pour chaque position $i \in \mathcal{D}_w$, $P_s(i)$ est vrai si et seulement si $w[i] = s$.

L'ensemble des mots finis sur un alphabet fini Σ correspond alors à l'ensemble de structures

$$\mathcal{S}_{\Sigma^*} = \bigcup_{w \in \Sigma^*} \mathcal{S}_w$$

Cette définition se généralise à la notion de structure sur les mots infinis en posant $\mathcal{D}^w = \mathbb{N}\backslash 0$ pour rester fidèle aux notations introduites dans le chapitre 2. L'ensemble des mots infinis sur un alphabet fini Σ correspond à l'ensemble de structures

$$\mathcal{S}_{\Sigma^\omega} = \bigcup_{w \in \Sigma^\omega} \mathcal{S}_w$$

La notion de structure sur les mots permet d'établir le lien entre logique du second ordre sur les entiers naturels et langages sur les mots finis. Pour un alphabet Σ, les structures $\mathcal{S}_w$ telles que $w \in \Sigma^\infty$ ont des domaines différents, mais utilisent toutes les même symboles de prédicats, bien que leur sémantique (interprétations) soient différentes. Ce concept est formellement capturé par la notion de **signature** [62]. Pour un ensemble de structures dotés de signatures identiques, il est naturel d'associer à la signature de l'ensemble la signature de n'importe quelle structure particulière de l'ensemble.

Théorème 1 (Logique et automate sur les mots finis [24] (1960))**.** *Soient Σ un alphabet et φ une formule de logique du second ordre monadique sur la signature de $\mathcal{S}_{\Sigma^*}$. Alors il est possible de construire un automate fini $\mathcal{A}$ d'alphabet Σ tel que l'ensemble des mots $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$ acceptés par $\mathcal{A}$ correspond à l'ensemble des modèles de φ de la façon suivante :*

$$\forall w \in \Sigma^*, w \in \mathcal{L}(\mathcal{A}) \Leftrightarrow \mathcal{S}_w \vDash \varphi$$

La construction inverse, qui consiste à spécifier un automate sur les mots finis par une formule logique est beaucoup plus classique, et est intégralement spécifiée dans [134].

Ce résultat invite naturellement à chercher une correspondance pour la logique sur les entiers naturels avec domaine $\mathcal{N}$, ce qui nécessite d'introduire la notion d'automates sur les mots infinis. Les automates de Büchi, présentés dans le chapitre 2, ont été introduits dans le but d'établir cette correspondance.

Théorème 2 (Logique et automate sur les mots infinis [26] (1962))**.** *Soient Σ un alphabet et φ une formule de logique du second ordre monadique sur la signature de $\mathcal{S}_{\Sigma^\omega}$. Alors il est possible de construire un automate de Büchi $\mathcal{A}$ d'alphabet Σ tel que l'ensemble des mots $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ acceptés par $\mathcal{A}$ correspond à l'ensemble des modèles de φ de la façon suivante :*

$$\forall w \in \Sigma^\omega, w \in \mathcal{L}(\mathcal{A}) \Leftrightarrow \mathcal{S}_w \vDash \varphi$$

La construction inverse, qui consiste à spécifier un automate sur les mots finis par une formule logique est similaire au cas des mots finis.[134].

Le rapprochement entre logique temporelle et informatique fut établi en 1977 lorsque A. Pnueli proposa d'utiliser la logique temporelle pour raisonner sur les programmes [104] (1977). L'approche était basée sur la théorie de preuve, ce qui ne nécessitait pas d'étudier plus en détail les procédures de satisfiabilité de la logique temporelle linéaire, mais nécessitait de construire des axiomatisation convenables pour les différentes logiques [44].

Le véritable intérêt pour les logiques temporelles provient de l'essor du *model checking* [31, 109] (1981 et 1982) pour vérifier systématiquement des programmes. Le problème de vérification en logique consiste à décider si une instanciation ou un ensemble d'instanciations donnés satisfait une formule logique. Ce problème prend un sens pratique lorsque l'ensembles d'instanciations à vérifier est l'ensemble des trace d'exécutions infinies d'un programme et que la formule logique est une spécification sur les traces d'exécution de ce programme. Les deux articles [31, 109] proposent une formalisation du problème utilisant des logiques temporelles arborescentes. La question du type de logique temporelle à utiliser a toujours été présente, mais il est maintenant établi que le type de logique la plus naturel à utiliser est une logique à structure temporelle linéaire [144].

En 1982, la situation était donc la suivante. D'une part, il était possible de décider la logique du second ordre monadique sur les entiers naturels avec ordre linéaire par des automates sur les

mots infinis, et d'autre part, il était possible de transformer la logique temporelle linéaire telle que définie par Kamp en une formule de logique du premier ordre sur les entiers naturels avec ordre linéaire. Enfin, il semblait tout à fait naturel d'appliquer les concepts du *model checking* au cas de la logique temporelle linéaire, ce qui pouvait se faire en transformant la spécification en formule du premier ordre, puis en convertissant la formule résultante en automate. Du fait de l'engouement et des résultats sus-cités, les progrès qui ont suivi ont été rapides, ce qui rend difficile d'établir une chronologie sur cette période.

Le problème de satisfaction pour la logique du premier ordre sur les entiers est non-élémentaire [130] (1974), alors que le problème de satisfaction pour la logique temporelle linéaire est PSPACE-complet [123] (1982). Ceci indique qu'il est préférable de transformer directement la spécification en automate [152] (1983). Une procédure simple pour transformer une formule de logique temporelle en automate est décrite dans [60] (1995), et un algorithme récent basé sur l'utilisation d'automates alternants faibles est décrit dans [56] (2001). Les automates alternants sont plus compact et permettent d'obtenir une procédure de transformation en temps linéaire [138], mais dans une problématique de synthèse de solutions, ils imposent de résoudre le problème de satisfaction propositionnelle au fur et à mesure du parcours de l'automate.

En terme d'expressivité, la logique temporelle linéaire est équivalente à la logique du premier ordre du successeur sur les entiers naturels [54], c'est à dire strictement moins expressive que la logique du second ordre monadique du successeur sur les entiers naturels capturée par les automates de Büchi. En terme de théorie des langages, la logique temporelle linéaire correspond aux langages *star-free* [103], alors que les automates de Büchi représentent les langages ω-réguliers. Il est à noter que toutes les conditions d'acceptations qui ont été définies sur les mots infinis produisent des automates dont les langages sont les langage ω-reguliers [48]. Il semble donc qu'il existe un analogue de la thèse de Church-Turing pour les langages ω-régulier, et il est alors naturel d'essayer d'étendre l'expressivité des logiques temporelles linéaires afin de couvrir l'intégralité des langages ω-réguliers.

Les propositions pour augmenter l'expressivité de la logique temporelle linéaire ont été multiples. La première consiste à augmenter la syntaxe avec une grammaire non contextuelle linéaire à droite, ce qui produit la logique ETL [151] (1983). Une autre proposition consiste à ajouter directement le support syntaxique pour spécifier des expression ω-régulières [142] (1994). La logique LTL quantifiée permet également d'atteindre l'ω-régularité [125] (1987). Enfin, une dernière possibilité consiste à utiliser les opérateurs point-fixes du μ-calcul [78] (1983) dans un contexte linéaire, pour obtenir la logique temporelle linéaire νTL [12] (1989). La logique μTL présentée dans [137] (1988), est une augmentation de la logique νTL avec des opérateurs pour exprimer le passé.

De nombreuses autres logiques temporelles ont été examinées comme candidates à la vérification de modèles, (cf. section 4.5). Le standard récent PSL [1] est le résultat de la sélection de quatre candidats proposés par Motorola, Intel, Verisity et IBM[139, 115].

— CBV (Motorola) [2] était basé sur le μ-calcul, avec le principe d'avoir l'apparence d'un langage de programmation plus que d'une logique ;

— Temporal **e** (Verisity) [27] est directement basé sur une généralisation de la logique temporelle linéaire utilisant des expressions régulières ;

— Sugar (IBM) [14] est basé sur la logique temporelle arborescente augmentée d'une syntaxe pour spécifier des expressions régulières ;

— ForSpec (Intel) [8] est basé sur la logique temporelle linéaire.

La conclusion du processus de standardisation a été une logique basée sur la logique temporelle linéaire qui utilise la syntaxe de Sugar pour la spécification des expressions régulières [115], ce qui montre qu'en ce qui concerne les applications du *model checking*, la logique temporelle linéaire sans point fixe semble pour le moment plus facile à utiliser que les autres logiques temporelles.

Pour finir, il est difficile d'évoquer le *model checking* sans mentionner l'usage fondamental qui est fait des diagrammes de décision binaire [23, 91] dans les algorithmes. Ceux-ci ont permis d'appliquer les techniques de *model checking* sur des systèmes extrêment grand [25], et commencent à être utilisés en programmation par contraintes [69]. Cependant, il s'agit de techniques d'optimisation qui, bien qu'elles soient très efficaces en pratique, n'ont pas d'influence sur la complexité intrinsèque des algorithmes ou sur l'expressivité des formalismes considérés.

Les algorithmes basés sur les automates proviennent du lien presque direct entre logique temporelle linéaire, logique du premier ordre et automates de Büchi, mais ces techniques ont été depuis appliquées à de nombreuses logiques apparentées qui sont décrites dans la section 4.5. Nous décrivons une construction particulière dans la section suivante, que nous utiliserons comme référence dans les contributions.

4.4 Logique temporelle linéaire

Dans cette section, nous décrivons en détail la logique propositionnelle temporelle linéaire LTL, ainsi qu'un algorithme de satisfaction basé sur la construction d'automates [60], et nous concluons sur l'adaptation de cet algorithme à la génération de solutions arbitraires pour le rapprocher du contexte de la satisfaction de contraintes.

La logique temporelle linéaire (LTL) est une logique modale qui permet d'exprimer des propriétés propositionnelles temporelles, et combine donc la logique propositionnelle avec des modalités temporelles. Les ensembles de modèles d'une formule LTL sont des langages sur les mots infinis d'alphabet les instanciations des variables propositionnelles de la formule. Cette notion est capturée par le concept d'interprétation temporelle.

Définition 51 (Interprétation temporelle)**.** Soient $\mathcal{D}$ le domaine d'une structure et $\mathcal{X}$ un ensemble de variables à valeurs dans $\mathcal{D}$. Une interprétation de X (Définition 45) est une fonction $I : X \to \mathcal{D}$, et l'ensemble des interprétations de X est noté $\mathcal{I}(X)$. Une **interprétation temporelle** de X est une fonction $I_T : \mathbb{N} \to \mathcal{I}(X)$ qui associe une instanciation des variables à chaque entier naturel. L'ensemble des interprétations temporelles de X est noté $\mathcal{I}_T(X)$. La restriction d'une instanciation temporelle à un sous-ensemble des variables est définie de façon analogue aux définitions antérieures.

Définition 52 (Décalage d'une interprétation temporelle)**.** Soient $\mathcal{D}$ le domaine d'une structure, $\mathcal{X}$ un ensemble de variables à valeurs dans $\mathcal{D}$ et $I \in \mathcal{I}_T(X)$ une interprétation temporelle de X. Nous notons $I[k, \omega]$ l'unique interpétation temporelle qui à tout $n \in \mathbb{N}$ et $x \in X$ associe la valeur $I[k, \omega](n)(x) = I(n + k)(x)$.

Propriété 4 (Sémantique de mot d'une interprétation temporelle)**.** *Soient $\mathcal{D}$ le domaine d'une structure et V un ensemble de variables à valeurs dans $\mathcal{D}$. L'ensemble des instanciations temporelles $\mathcal{I}_T(\mathcal{V})$ est isomorphique à l'ensemble des mots infinis $\mathcal{I}(\mathcal{D})^\omega$ via le morphisme qui à $I \in \mathcal{I}_T(\mathcal{V})$ associe le mot $w \in \mathcal{I}(\mathcal{D})^\omega$ tel que :*

$$\forall n > 0, \forall v \in \mathcal{V}, w[n](v) = I(n-1)(v)$$

La sémantique de mot permet d'établir la correspondance entre logique LTL et automates.

4.4.1 Syntaxe et sémantique

La syntaxe de la logique LTL est constituée des connecteurs de la logique propositionnelle, ainsi que de connecteurs temporels. L'algorithme que nous décrirons exige que les formules soient présentées sous une forme normale qui utilise seulement les connecteurs propositionnels $\{\wedge, \vee, \neg\}$ et les connecteurs temporels $\{\mathbf{X}, \mathbf{U}, \mathbf{V}, \mathbf{F}, \mathbf{G}\}$ dont les sémantiques sont données ci-dessous. De nombreuses variantes de la logique LTL existent [44]. Le connecteur **F** est omis dans [60], car il peut-être spécifié à partir du connecteur '**U**', mais il est si fondamental que nous préférons l'inclure dans l'algorithme. Le connecteur '**V**'est défini spécifiquement dans [60] pour rendre l'existence d'une forme normale possible, est correspond au dual de '**U**'. Une variante du connecteur '**V**'est le connecteur '**R**' défini dans [138]. Enfin, le passé '**S**' n'est pas habituellement pris en compte dans la logique LTL, et son inclusion permet de gagner en concision, mais ne permet pas de gagner en expressivité [83].

Syntaxe de LTL. Soient $\mathcal{P}$ un ensemble dénombrable de variables propositionnelles. La **syntaxe** de la logique LTL est définie inductivement de la façon suivante :

- Une variable propositionnelle $p \in \mathcal{P}$ est une formule
- Si φ est une formule, alors $\neg\varphi$ est une formule
- Si φ et θ sont des formules, alors $\varphi \wedge \theta$ et $\varphi \vee \theta$ sont des formules
- Si φ est une formule, alors $\mathbf{X}\varphi$, $\mathbf{G}\varphi$, $\mathbf{F}\varphi$ sont des formules
- Si φ et θ sont des formules, alors $\varphi\mathbf{U}\theta$ et $\varphi\mathbf{V}\theta$ sont des formules

De même qu'en logique propositionnelle, l'ensemble de variables $\mathrm{var}(\varphi) \subset \mathcal{P}$ d'une formule LTL φ est constitué des variable propositionnelles présentes dans φ.

Semantique de LTL La sémantique de la logique LTL est considérée par rapport à des interprétations temporelles :

- $I \vDash p$ si et seulement si $I(0)(p) = \top$,
  où $p \in \mathcal{I}_T(\{p\})$;
- $I \vDash \neg\varphi$ s'il n'est pas le cas que $I \vDash \varphi$,
  où $I \in \mathcal{I}(\mathrm{var}(\varphi))$;
- $I \vDash \psi \wedge \theta$ si et seulement si $(I{\downarrow}\mathrm{var}(\psi)) \vDash \psi$ et $(I{\downarrow}\mathrm{var}(\theta)) \vDash \theta$,
  où $I \in \mathcal{I}_T(\mathrm{var}(\psi) \cup \mathrm{var}(\theta))$;
- $I \vDash \psi \vee \theta$ si et seulement si $(I{\downarrow}\mathrm{var}(\psi)) \vDash \psi$ ou $(I{\downarrow}\mathrm{var}(\theta)) \vDash \theta$,
  où $I \in \mathcal{I}_T(\mathrm{var}(\psi) \cup \mathrm{var}(\theta))$;
- $I \vDash \mathbf{X}\varphi$ si et seulement si $I[1, \omega] \vDash \varphi$,
  où $I \in \mathcal{I}_T(\mathrm{var}(\varphi))$;
- $I \vDash \mathbf{G}\varphi$ si et seulement si pour tout $k \geqslant 0$, $I[k, \omega] \vDash \varphi$,
  où $I \in \mathcal{I}_T(\mathrm{var}(\varphi))$;
- $I \vDash \mathbf{F}\varphi$ si et seulement s'il existe $k \geqslant 0$ tel que $I[k, \omega] \vDash \varphi$,
  où $I \in \mathcal{I}_T(\mathrm{var}(\varphi))$;
- $I \vDash \psi\mathbf{U}\theta$ si et seulement s'il existe $k \geqslant 0$ tel que pour tout $j < k$, $(I{\downarrow}\mathrm{var}(\psi))[j, \omega] \vDash \psi$, et $(I{\downarrow}\mathrm{var}(\theta))[k, \omega] \vDash \theta$, où $I \in \mathcal{I}_T(\mathrm{var}(\psi) \cup \mathrm{var}(\theta))$;

— $I \vDash \psi \mathbf{V} \theta$ si et seulement si pour tout $j \geqslant 0$, si $(I{\downarrow}\mathrm{var}(\theta))[j, \omega] \vDash \theta$, alors il existe $k < j$, telle que $(I{\downarrow}\mathrm{var}(\psi))[k, \omega] \vDash \psi$, où $I \in \mathcal{I}_T(\mathrm{var}(\varphi))$.

Il sera utile dans la suite de travailler avec les mots infinis sur les sous-ensembles de littéraux.

Définition 53 (Mots infinis sur les littéraux)**.** Les mots infinis sur les sous-ensembles de littéraux permettent de définir concisément des ensemble d'interprétations propositionnelles. À un mot $w \in \mathrm{Part}(L(\mathcal{P}))$, nous faisons correspondre l'ensemble d'instanciations temporelles $I_T(w)$ suivant :

$$I_T(w) = \{I \in \mathcal{I}_T(\mathcal{P}) \mid \forall n > 0, \forall p \in \mathcal{P}, p \in w[n] \rightarrow I(n)(p) = \top \wedge \neg p \in w[n] \rightarrow I(n)(p) = \bot\}$$

Forme normale négative Une formule LTL φ présentée selon la syntaxe ci-dessus peut être mise sous forme normale négative en utilisant les équivalences suivantes de gauche à droite. Le résultat est une expression dans laquelle l'opérateur de négation '$\neg$' figure seulement devant des variables propositionnelles.

— $\neg(\mathbf{X}\varphi) \equiv \mathbf{X}(\neg\varphi)$
— $\neg(\mathbf{F}\varphi) \equiv \mathbf{G}(\neg\varphi)$
— $\neg(\mathbf{G}\varphi) \equiv \mathbf{F}(\neg\varphi)$
— $\neg(\varphi \mathbf{U} \theta) \equiv (\neg\varphi)\mathbf{V}(\neg\theta)$
— $\neg(\varphi \mathbf{V} \theta) \equiv (\neg\varphi)\mathbf{U}(\neg\theta)$
— $\neg(\varphi \vee \theta) \equiv (\neg\varphi) \wedge (\neg\theta)$
— $\neg(\varphi \wedge \theta) \equiv (\neg\varphi) \vee (\neg\theta)$

4.4.2 Automate de Büchi généralisé arborescent

L'algorithme de [60] effectue des copies d'états partiels qui masquent la structure arborescente de la construction. Nous définissons une variante de la notion d'automate de Büchi dans laquelle les états sont des arbres, et nous présentons une variante de l'algorithme de [60] qui utilise ces automates.

Définition 54 (Automate de Büchi généralisé arborescent)**.** Soit φ une formule de logique LTL. Un **automate de Büchi généralisé arborescent** (ABGA) associé à φ est un automate $\mathcal{A}_\varphi = \langle Q, Q_0, \Sigma, \delta, F\rangle$ dans lequel :
— Q est un ensemble d'arbres ;
— $Q_0 \subseteq Q$ est l'ensemble des états initiaux ;
— $\Sigma = \mathrm{Part}(L(\mathrm{var}(\varphi))$;
— $\delta : \mathrm{NOEUDS}_\forall(Q) \times \Sigma \rightarrow Q$ est une fonction de transition déterministe ;
— $\mathcal{F} \in \mathrm{Part}(\mathrm{NOEUDS}_\forall(Q))^*$ est une condition de Büchi généralisée sur les nœuds des arbres.

Acceptation d'un mot par un automate de Büchi généralisé arborescent. Un mot $w \in \Sigma^\omega$ est accepté par $\mathcal{A}_\varphi$ s'il existe une séquence infinie de nœuds $\xi \in \mathrm{NOEUDS}_\forall(\mathcal{A})^\omega$ qui satisfait les conditions suivantes :
— $\mathrm{GRAPHE}(\xi[1]) \in Q_0$
— Pour tout $k > 0$, $\mathrm{GRAPHE}(\delta(\xi[k], w[k])) = \mathrm{GRAPHE}(\xi[k+1])$
— Pour tout $F \in \mathcal{F}$, $\mathrm{Inf}(\xi) \cap F \neq \emptyset$

Enfin, une interprétation temporelle $I \in \mathcal{I}_T(\mathrm{var}(\varphi)$ est acceptée par $\mathcal{A}_\varphi$ si et seulement si il existe un mot $w \in \Sigma^\omega$ tel que $w \in \mathcal{I}_T(w)$.

Correspondance avec les automates de Büchi généralisés. Un automate de Büchi généralisé arborescent $\mathcal{A}_\varphi = \langle Q, Q_0, \Sigma, \delta, F\rangle$ correspond à l'automate de Büchi généralisé $\mathcal{A}^\times_\varphi = \langle Q^\times, Q^\times_0, \Sigma^\times, \delta^\times, \mathcal{F}^\times\rangle$ construit de la façon suivante :

— $Q^\times = \text{NOEUDS}_\forall(Q)$

— $Q^\times_0 = \{n \in \text{NOEUDS}_\forall(Q) \mid \text{GRAPHE}(n) \in Q_0\}$

— $\Sigma = \text{Part}(\text{var}(\varphi))$

— Pour tout $q \in Q^\times$ et $s^\times \in \Sigma^\times$, $\delta^\times(q, s^\times) = \{q' \in Q^\times \mid \delta(q, s) = \text{GRAPHE}(q') \wedge I(s^\times) \vDash \varphi(s)\}$ où $\varphi(s) = \bigwedge_{\ell \in s} \ell$ et $I(s^\times)$ est l'interprétation $I \in \mathcal{I}_P(\text{var}(\varphi))$ telle que pour tout $v \in \text{var}(\varphi)$, $I(v) = \top$ si et seulement si $v \in s^\times$, .

— $\mathcal{F}^\times = \mathcal{F}$

Intuitivement, alors que dans l'automate de Büchi généralisé arborescent, les transitions se font des feuilles d'un arbres vers les racines d'autres arbres, dans l'automate de Büchi généralisé correspondant, les transitions se font directement vers l'ensemble des feuilles de l'arbre correspondant. Également, les automates de Büchi généralisés considèrent traditionnellement des sous-ensembles de variables qui représentent des instanciations complètes, alors qu'il est plus compact de manipuler des ensembles de littéraux qui représententent des ensembles d'instanciations. Un exemple figure dans la section 4.4.4.

4.4.3 Algorithme

Le principe de l'algorithme consiste à construire un automate de Büchi généralisé arborescent $\mathcal{A}_\varphi$ associé à une formule LTL φ mise sous forme normale négative, pour lequel le langage $\mathcal{L}(\mathcal{A}_\varphi) \subseteq \text{Part}(\text{var}(\varphi))^\omega$ est précisément l'ensemble des mots infinis qui sont des modèles de φ.

Les états de $\mathcal{A}_\varphi$ sont des arbres dont les nœuds sont des ensembles de formules LTL. La construction est incrémentale, et un ensemble de formules LTL conduit à la génération d'un arbre dont les feuilles conduisent à la génération d'autres états.

L'algorithme est basé sur les équivalences point-fixe suivantes pour les opérateurs '**U**' et '**V**' :

— $\varphi \mathbf{U} \theta \equiv \theta \vee (\varphi \wedge \mathbf{X}(\varphi \mathbf{U} \theta))$

— $\varphi \mathbf{V} \theta \equiv (\theta \wedge \varphi) \vee (\theta \wedge \mathbf{X}(\varphi \mathbf{V} \theta))$

Les opérateurs '**G**' et '**F**' sont des formes spécifiques de '**U**' et '**V**', car $\mathbf{F}\varphi \equiv \top \mathbf{U} \varphi$ et $\mathbf{G}\varphi \equiv \varphi \mathbf{V} \bot$. Les équivalences associés à '**F**' et '**G**' sont donc des simplifications des équivalences précédentes.

— $\mathbf{F}\varphi \equiv \varphi \vee \mathbf{X}\mathbf{F}\varphi$

— $\mathbf{G}\varphi \equiv \varphi \wedge \mathbf{X}\mathbf{G}\varphi$

Intuitivement, le principe consiste à considérer une formule LTL comme une conjonction de formules qui doivent être satisfaites à l'instant initial. Si une variable propositionnelle p doit être satisfaite, alors la première interprétation doit associer $\top$ à p. Si une formule de la forme $\mathbf{X}\varphi$ doit être satisfaite à l'instant initial, alors φ doit être satisfaite à l'instant suivant. Les équivalences précédentes permettent de décomposer les opérateurs '**U**', '**V**', '**F**' et '**G**' en une composante qui doit être satisfaite à l'instant initial et une composante qui doit être satisfaite à l'instant suivant.

L'algorithme 4.1 prend en entrée une formule LTL φ mise sous forme normale négative (type `LTLNNF`), et produit l'automate de Büchi généralisé arborescent $\mathcal{A}_\varphi$ associé à φ (type `ABGA`). La fonction `Initialiser` (Algorithme 4.2) initialise l'automate. La fonction `Étendre` (Algorithme

4.3) ajoute à un ensemble de formules LTL toutes des formules qui sont des conséquences directes. La fonction `CreerEnfants` (Algorithme 4.4) produit l'arbre associé à un ensemble de formules, et la fonction `CreerEtats` (Algorithme 4.5) produit les états engendrés par les feuilles d'un état. Une fois que tous les états ont été créés, la fonction `CreerTransitions` (Algorithme 4.6) créer les transitions, puis la fonction `ConditionAcceptation` (Algorithme 4.7) génère la condition d'acceptation.

Le type $\mathtt{T}_{\mathtt{Ens}_{\mathtt{LTL}}}$ représente les arbres dont les nœuds sont des ensembles de formules LTL.

Dans la suite, nous utiliserons la fonction $\text{Etiquette}_{\cup}(T, n)$ pour dénoter l'union des étiquettes des ancêtres d'un nœud n d'un arbre T. Cette fonction n'est pas introduite dans le chapitre 2 car elle est spécifique aux arbres dont les étiquettes des nœuds sont des ensembles, et est seulement utilisée dans cette section.

$$\text{Etiquette}_{\cup}(T, n) = \bigcup_{a \in \text{Ancetres}(n,T)} \text{Etiquette}(T, a)$$

Algorithme 4.1 Construction de $\mathcal{A}_\varphi$

1: **procédure** `Generer`(φ : `LTLNNF`) : `ABGA`
2: $\mathcal{A}_\varphi$: `ABGA`
3: `Initialiser`($\varphi, \mathcal{A}_\varphi$)
4: $\mathcal{T}_\varphi$: $\mathtt{T}_{\mathtt{Ens}_{\mathtt{LTL}}}(\{\varphi\})$
5: `Étendre`(`Etiquette`($\mathcal{T}_\varphi$))
6: `CreerEnfants`($\mathcal{T}_\varphi$, `Racine`($\mathcal{T}_\varphi$))
7: `CreerEtats`($\mathcal{A}_\varphi, \mathcal{T}_\varphi$)
8: `CreerTransitions`($\mathcal{A}_\varphi, \varphi$)
9: `ConditionAcceptation`($\mathcal{A}_\varphi, \varphi$)
fin procédure

Initialisation. L'initialisation de l'automate impose seulement l'alphabet. Tout le reste sera produit par d'autres procédures.

Algorithme 4.2 Initialisation de l'automate de Büchi généralisé $\mathcal{A}_\varphi$

1: **procédure** `Initialiser`(φ : `LTLNNF`, ! $\mathcal{A}$: `ABG`)
2: `Alphabet`($\mathcal{A}$) = `Part`(L(`var`(φ)))
3: `Etats`($\mathcal{A}$) = $\emptyset$
4: `Initiaux`($\mathcal{A}$) = $\emptyset$
5: `Transitions`($\mathcal{A}$) = $\emptyset$
6: `BuchiG`($\mathcal{A}$) = $\emptyset$
fin procédure

Extension. L'objectif de l'algorithme 4.3 est d'ajouter à un ensemble de formules les termes de toutes les conjonctions dudit ensemble, ainsi que les formules issues d'une équivalence point-fixe. Ceci permet d'obtenir un ensemble de formules pour lequel le non-déterministe est explicitement

et uniquement présent dans les disjonctions, et cela permet également de construire des ensemble normalisés de formules qui peuvent servir d'identifiants pour la fonction de transition.

La boucle provient du fait que les formules ajoutées peuvent elles-mêmes être des conjonctions. Les opérateurs '**F**' et '**U**' sont également traités ici afin que l'algorithme `CreerEnfants` (Algorithme 4.4) n'ait à s'occuper que des disjonctions.

Algorithme 4.3 Extension d'un ensemble de formules par les formules implicites

```
1:  procédure Étendre(! E : Ens_LTL)
2:    t := |n|
3:    tant que ⊤ faire
4:      pour chaque Gφ ∈ E faire
5:        Ajouter(E, φ ∧ XGφ)
6:      fin pour
7:      pour chaque Fφ ∈ E faire
8:        Ajouter(E, φ ∨ X(Fφ))
9:      fin pour
10:     pour chaque φUθ ∈ E faire
11:       Ajouter(E, θ ∨ (φ ∧ X(φUθ)))
12:     fin pour
13:     pour chaque φVθ ∈ E faire
14:       Ajouter(E, (θ ∧ φ) ∨ (θ ∧ X(φVθ)))
15:     fin pour
16:     pour chaque φ ∧ θ ∈ E faire
17:       Ajouter(E, φ)
18:       Ajouter(E, θ)
19:     fin pour
20:     si |E| = t alors
21:       Sortir
22:     fin si
23:     t :=|E|
24:   fin tant que
    fin procédure
```

Création de nœuds enfants La fonction `CreerEnfants` de l'algorithme 4.4 gère le non-déterminisme en associant à chaque disjonction $\varphi \vee \theta$ un nœud pour φ et un nœud pour θ. S'il existe une disjonction parmi les formules des ancêtres d'un nœud qui n'a pas été traitée, alors un nœud est créé pour chaque terme, et la procédure est réitérée sur ces nouveaux nœuds jusqu'à ce que toutes les disjonctions aient été traitées.

Création des états. La fonction `CreerEtats` de l'algorithme 4.5 consiste à collecter l'ensemble des formules de la forme $\mathbf{X}\varphi$ qui doivent être satisfaites à l'instant suivant, et à produire l'ensemble de formules correspondant. L'ensemble de formules obtenu est normalisé par la procédure `Etendre` afin de mieux correspondre à un identificateur d'état. Cette procédure fait appel à elle même pour créer autants d'états que nécessaire en examinant les nœuds feuille de chaque nouvel état créé.

Algorithme 4.4 Développement de l'arbre

```
1: procédure CreerEnfants(!𝒯 : T_LTL, n : ℕ)
2:   pour chaque φ ∨ θ ∈ Etiquette_∪(𝒯, n) faire
3:     pour chaque ψ ∈ {φ, θ} faire
4:       si ψ ∉ F alors
5:         n' = CreerNoeud(𝒯, {ψ})
6:         AjouterEnfant(𝒯, n, n')
7:         Etendre(Etiquette(𝒯, n'))
8:         CreerEnfants(𝒯, n')
9:       fin si
10:    fin pour
11:    sortir
12:  fin pour
   fin procédure
```

Algorithme 4.5 Création des états

```
1: procédure CreerEtats(!𝒜 : ABG, 𝒯 : T_Ens_LTL)
2:   pour chaque f ∈ Feuilles(𝒯) faire
3:     F = Etendre({φ | Xφ ∈ Etiquette_∪(𝒯, f)})
4:     si ∀Q ∈ Etats(𝒜), Etiquette(Q) ≠ F alors
5:       𝒯_F : T_LTL(F)
6:       Ajouter(Etats(𝒜), 𝒯_F)
7:       CreerEnfants(𝒯_F, Racine(𝒯_F))
8:       CreerEtats(𝒜, 𝒯_F)
9:     fin si
10:  fin pour
   fin procédure
```

Création des transitions Dans cette présentation, les transitions sont créées une fois que tous les états ont été créés. Dans le cas d'un algorithme en ligne, il est possible d'intercaler la création de transitions avec la créations d'états, et de ne mémoriser que les racines des états créés pour les états dont il a été montré que tous les chemins sont infructueux. La dissociation que nous effectuons ici implique de mémoriser l'automate dans sont intégralité.

La procédure `CreerTransitions` de l'algorithme 4.6 consiste à collecter pour chaque nœud feuille de chaque état, l'ensembles des formules de la forme $\mathbf{X}\varphi$ afin de déterminer l'état suivant, et l'ensembles des littéraux afin de déterminer les conditions de la transition, c'est-à-dire les symboles de l'alphabet à examiner pour cette transition.

Algorithme 4.6 Création des transitions

```
1: procédure CreerTransitions(!A : ABG, φ : LTLNNF)
2:   pour chaque Q ∈ Etats(A) faire
3:     pour chaque f ∈ Feuilles(Q) faire
4:       F = Etendre(Etiquette_∪(Q, f))
5:       F_X = {ψ | Xψ ∈ F}
6:       F_P = F ∩ (⋃_{p∈var(φ)} {p, ¬p})
7:       pour chaque T ∈ {Q ∈ Etats(A) | Etiquette(Q) = F_X} faire
8:         AjouterTransition(A, ⟨Q, f⟩, F_P, T)
9:       fin pour
10:    fin pour
11:  fin pour
   fin procédure
```

Condition d'acceptation La partie la moins intuitive de l'algorithme est la condition d'acceptation, décrite dans la procédure `ConditionAcceptation` de l'algorithme 4.7. Le principe consiste, étant donnée une formule $\psi\mathbf{U}\theta \equiv \theta \vee (\psi \wedge \mathbf{X}(\psi\mathbf{U}\theta))$, à contraindre l'automate à finir par entrer dans un état satisfaisant θ, c'est-à-dire à empêcher les boucles infinies sur ψ. Afin de ne pas impacter les autres états satisfaisants θ mais pas $\psi\mathbf{U}\theta$, cette condition est exprimée par une disjonction exclusive qui inclus les états qui satisfont θ mais pas $\mathbf{X}(\psi\mathbf{U}\theta)$.

4.4.4 Exemple

Nous illustrons l'algorithme sur la formule $\varphi = (p\mathbf{U}q) \vee (\mathbf{G}p \wedge \mathbf{F}\neg p)$. Dans cette formule, le terme $p\mathbf{U}q$ de la disjonction est satisfaisable, mais le terme $\mathbf{G}p \wedge \mathbf{F}\neg p$ ne l'est pas car il est impossible d'avoir à la fois "toujours p" et de finir par avoir $\neg p$ sur un chemin donné.

Pour un ensemble de formules Φ, nous notons $\mathcal{T}(\Phi)$ l'état Etendre(Φ) associé. Par exemple, l'état associé à $\{\mathbf{G}p\}$ est l'état dont la racine est étiquetée par l'ensemble de formules $\{\mathbf{G}p, p, \mathbf{XG}p\}$.

La procédure `Etendre` appliqué sur $\{\varphi\}$ n'a aucun effet, car φ est une disjonction. Le premier appel à `CreerEnfants` produit des nœuds étiquetés par les ensembles $\{p\mathbf{U}q\}$ et $\{\mathbf{G}p \wedge \mathbf{F}\neg p\}$. La procédure `Etendre` appliquée à ce dernier produit l'ensemble $\{\mathbf{G}p \wedge \mathbf{F}\neg p, \mathbf{G}p, \mathbf{F}\neg p, p, \mathbf{XG}p\}$. Le produit final du premier appel à `CreerEnfants` et l'arbre de la figure 4.3. Les numéros des nœuds feuille sont indiqués en dessous à droite de ceux-ci. Par exemple, le nœud dont l'étiquette est $\{q\}$ porte le numéro 1. Les transitions sont indiquées par des flèches étiquetés par des ensembles

Algorithme 4.7 Mise en place de la condition d'acceptation

procédure `ConditionAcceptation`($!\mathcal{A} : \texttt{ABG}, \varphi : \texttt{LTLNNF}$)
  $F = \bigcup_{Q \in \text{Etats}(\mathcal{A}), n \in \text{Noeuds}(Q)} \text{Etiquette}(Q, n)$
  pour chaque $\mu\mathbf{U}\psi \in F$ **faire**
    $F_{\mu\mathbf{U}\psi} = \{\langle q, n\rangle \in \text{NOEUDS}_\forall(\mathcal{A}) \mid (\mu\mathbf{U}\psi \notin \texttt{Etiquette}_\cup(q,n)) \oplus (\psi \in \texttt{Etiquette}_\cup(q,n))\}$
    `Ajouter`($\texttt{BuchiG}(\mathcal{A}), F_{\mu\mathbf{U}\psi}$)
  fin pour
  pour chaque $\mathbf{F}\psi \in F$ **faire**
    $F_{\mu\mathbf{U}\psi} = \{\langle q, n\rangle \in \text{NOEUDS}_\forall(\mathcal{A}) \mid (\mathbf{F}\psi \notin \texttt{Etiquette}_\cup(q,n)) \oplus (\psi \in \texttt{Etiquette}_\cup(q,n))\}$
    `Ajouter`($\texttt{BuchiG}(\mathcal{A}), F_{\mathbf{F}\psi}$)
  fin pour
fin procédure

de littéraux, et dont les extrémités pointent vers les autres états, qui sont symbolisés par des rectangles aux coins arrondis.

L'algorithme que nous décrivons ici ne tente pas d'identifier les états inconsistants, c'est-à-dire ceux qui contiennent à la fois une variable propositionnelle p et sa négation $\neg p$. Une optimisation consisterait à identifier et supprimer ces nœuds au fur et à mesure. Dans le cas présent, les nœuds $\langle \mathcal{T}(\varphi), 3\rangle$ et $\langle \mathcal{T}(\{\mathbf{G}p, \mathbf{F}\neg\, p\}, 1\rangle$ sont inconsistants, et leur suppression aurait évité la production de $\mathcal{T}(\{\mathbf{G}p\})$.

Discussion de la condition d'acceptation. La condition d'acceptation est donnée dans la figure 4.5. Dans l'article original [60], les auteurs indiquent qu'un automate de Büchi avec condition d'acceptation généralisée est équivalent à un automate de Büchi avec une condition d'acceptation simple comportant $n \times k$ états, où n est le nombre de nœud de l'automate généralisé initial et k la cardinalité de la condition d'acceptation généralisée.

Seuls les formules de la forme $\psi\mathbf{U}\theta$ conduisent à la création d'une condition d'acceptation généralisée. Le principe consiste à éviter les boucles infinies sur ψ, ce qui implique de "forcer" l'automate à entrer dans un état satisfaisant θ en incluant les états concernant θ. Cependant, bien qu'il est possible qu'un chemin infini passe plusieurs fois par θ, seul la première occurence de ce passage est nécessaire, et aucun état successeur ne mentionnera plus $\psi\mathbf{U}\theta$ dans ses conditions.

Sans la modalité '**G**', un chemin satisfaisant une formule LTL finira nécessairement dans l'état $\mathcal{T}(\emptyset)$, et il suffit donc seulement d'inclure $\langle \mathcal{T}(\emptyset), r\rangle$ dans la condition d'acceptation. Pour traiter de la modalité '**G**', nous suspectons qu'il est possible d'établir une condition plus simple qui permettrait d'obtenir un automate de Büchi doté d'un unique ensemble d'état finaux, ce qui éviterait ce malencontreux écart de taille entre les deux automates.

Problème de synthèse. Les automates associés aux logiques temporelles sont principalement utilisés dans le contexte du *model checking*, dans lequel le problème consiste à vérifier qu'une implémentation I, donnée sous la forme d'un automate A_I, satisfait une formule φ. Une méthode consiste à construire l'automate $A_I \cap A_{\neg\varphi}$, de façon explicite ou implicite, et à vérifier que le langage de celui-ci n'est pas vide, opération qui ne nécessite qu'un parcours en profondeur de l'automate.

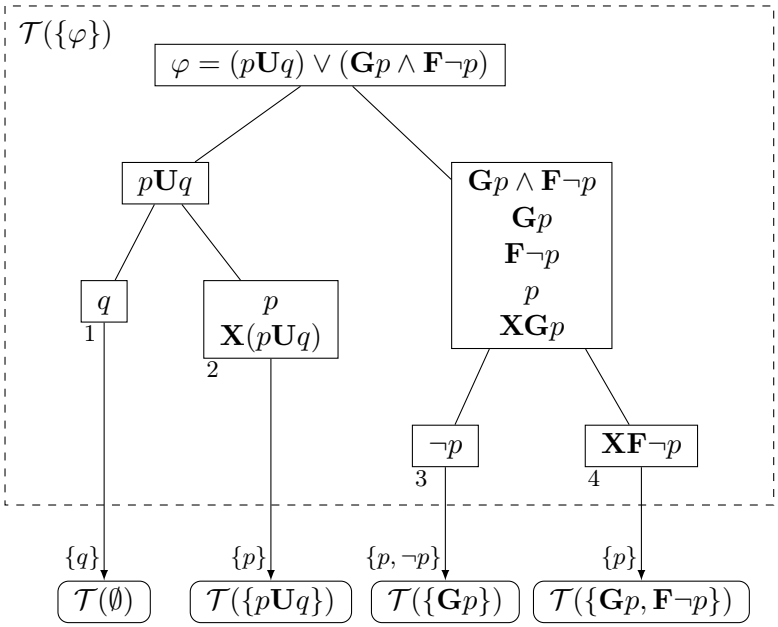

FIGURE 4.3 – Arbre $\mathcal{T}(\{\varphi\})$ de l'automate de Büchi arborescent généralisé associé à φ

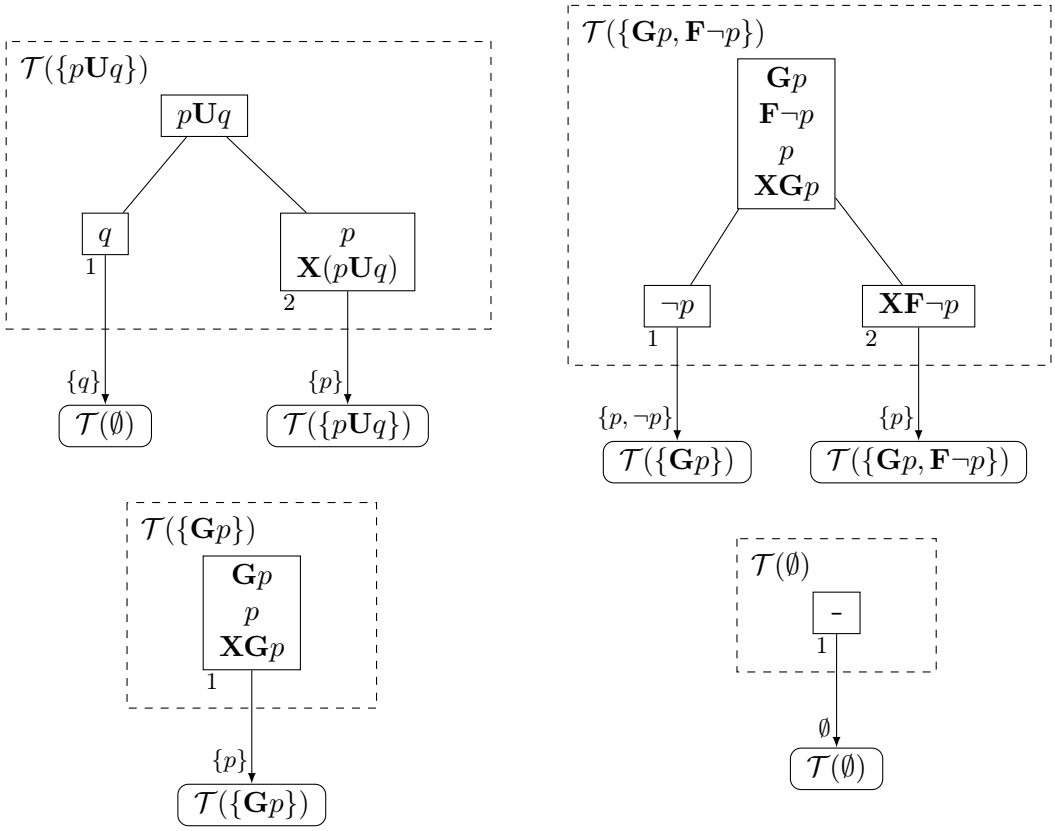

FIGURE 4.4 – Arbres $\mathcal{T}(\{p\mathbf{U}q\})$, $\mathcal{T}(\{\mathbf{G}p\})$, $\mathcal{T}(\{\mathbf{G}p, \mathbf{F}\neg p\})$ et $\mathcal{T}(\emptyset)$ de l'automate de Büchi arborescent généralisé associé à φ

$$\mathcal{F}_{p\mathbf{U}q} = \text{NOEUDS}_\forall(\mathcal{A}) \backslash \{\langle \mathcal{T}(\{\varphi\}), 2\rangle, \langle \mathcal{T}(\{p\mathbf{U}q\}), 2\rangle\}$$
$$\mathcal{F}_{\mathbf{F}\neg p} = \text{NOEUDS}_\forall(\mathcal{A}) \backslash \{\langle \mathcal{T}(\{\varphi\}), 4\rangle, \langle \mathcal{T}(\{\mathbf{G}p, \mathbf{F}\neg p\}), 2\rangle\}$$

FIGURE 4.5 – Condition d'acceptation de l'automate de Büchi généralisé arborescent $\mathcal{A}_\varphi$ associé à la formule $\varphi = (p\mathbf{U}q) \vee (\mathbf{G}p \wedge \mathbf{F}\neg p)$

Le présent exemple contient l'alternative $\mathbf{G}p \wedge \mathbf{F}\neg p$ qui est insatisfaisable. Les méthodes de *model checking* se contentent de chercher des chemins qui relient un état initial à un état final, et deux états finaux entre eux, ou une séquence d'états finaux dans le cas de la condition de Büchi généralisée.

Dans un contexte de génération, il serait intéressant de parcourir l'automate de façon arbitraire afin de produire une solution. Une telle procédure devra éviter les états inconsistants ainsi que les boucles infinies insatisfaisables, comme $\mathcal{T}(\{\mathbf{G}p, \mathbf{F}\neg p\}$, ce qui impose de détecter et supprimer les composantes fortement connexes de l'automate susceptibles de mener une procédure de génération vers des situations d'interblocage actif. L'idéal serait qu'une telle procédure puisse générer des nouveaux états rapidement, et la nécessité de rebrousser chemin au cours de la génération devrait alors être évitée.

La détection des composantes fortement connexes nécessite de travailler sur l'intégralité de l'automate. Une composante fortement connexe peut être éliminée de l'automate si et seulement si il s'agit d'une feuille dans le graphe des composantes fortement connexes et qu'elle ne contient aucun état finaux. Dans ce cas, la composante parent peut devenir une feuille, et devra alors être examinée. Les composantes fortement connexes singulières qui sont des feuilles doivent toujours être éliminées.

4.5 Autres logiques

La logique temporelle linéaire LTL s'inscrit dans un ensemble de logiques plus large, que nous détaillons dans cette section.

La logique LTL fait partie des logiques temporelles linéaires [44], pour lesquelles la complexité du problème de satisfaction varie en fonction des modalités temporelles **F**, **G**, **X**, **U**, **S** autorisées [123, 111]. La complexité PSPACE du problème de satisfaction pour la logique LTL découle principalement de la modalité **U**, en particulier parce qu'une version de cet opérateur permet de définir tous les autres [44]. La modalité temporelle **S** permet d'ajouter la possibilité de référer à des instants antérieurs, ce qui permet de gagner en concision, mais ne change pas l'expressivité ni la complexité du problème de satisfaction [83]. Les extensions de la logique LTL aux langages ω-reguliers ajoutent en expressivité, mais ne changent pas la complexité du problème de satisfaction [137].

Le cousin de la logique LTL en vérification des modèles est la logique temporelle arborescente CTL [31], qui ajoute la quantification universelle et existentielle sur les chemins, avec la restriction syntaxique que tout opérateur modal soit nécessairement précédé d'un quantificateur. Ceci implique que les modèles de la logique CTL ne sont pas des mots infinis, mais des langages sur les mots infinis. Par exemple, la formule $\varphi = \mathbf{E}p \wedge \mathbf{E}\neg p$ décrit l'ensemble des langages dans lesquels il existe un mot infini le long duquel p est vrai au premier instant, ainsi qu'un autre mot infini le long duquel p n'est pas vrai au premier instant. Comme un mot infini sur $w \in \text{Part}(\{p\})^\omega$

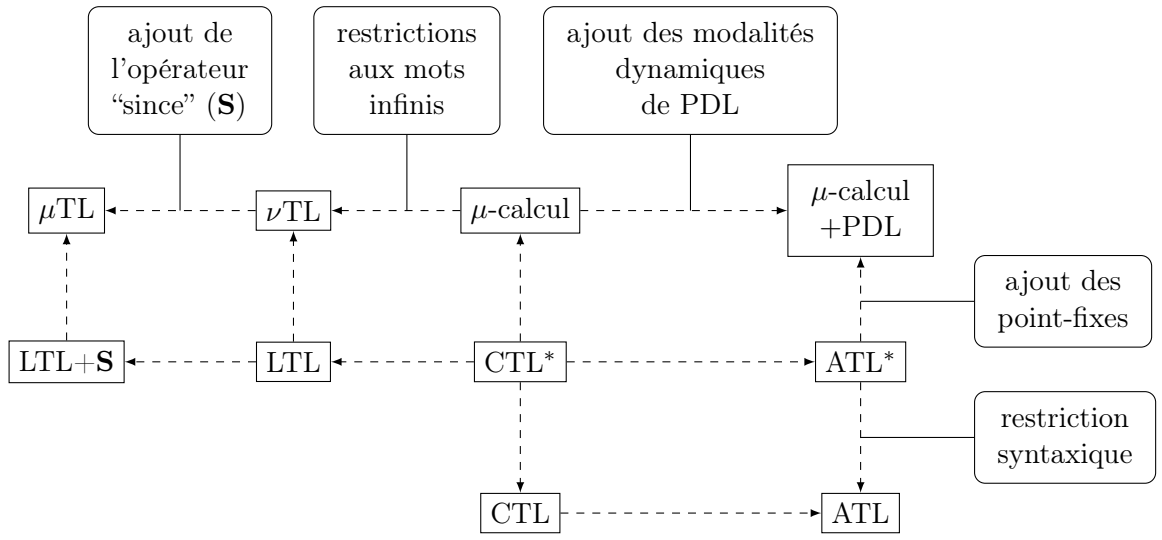

FIGURE 4.6 – Liens de parentés entre quelques logiques temporelles

ne peut satisfaire à la fois $p \in w[1]$ et $p \notin w[1]$, un modèle de φ doit être constitué d'au moins deux mots infinis. Les expressivités de LTL et CTL sont incomparables [81], et les complexités des algorithmes associés sont différentes. Pour la logique LTL, le problème de satisfaction et le problème de vérification sont tous les deux PSPACE-complet [124], et pour la logique CTL, le problème de satisfaction est EXPTIME-complet [45], alors que le problème de vérification est NLOGSPACE-complet [79]. La facilité du problème de vérification est ce qui a rendu initialement la logique CTL attractive pour le *model checking*, mais de nombreux arguments rendent la logique LTL préférable à la logique CTL en pratique [115].

La logique CTL* est le résultat de l'unification des logiques LTL et CTL, qui consiste à supprimer la restriction sur l'emplacement des quantificateurs [46]. Par exemple, la formule $\mathbf{AX}(\varphi \mathbf{U} \psi)$ est une formule de CTL* qui n'est pas une formule de CTL, car l'opérateur $\mathbf{U}$ est précédé de l'opérateur $\mathbf{X}$ qui n'est pas un quantificateur. La logique CTL* englobe la logique LTL en associant à toute formule φ de LTL la formule $E\varphi$ de CTL*. Pour cette logique, le problème de satisfaction est 2EXPTIME-complete [140] et le problème de vérification est PSPACE-complet [47]. Malgré une complexité similaire à celle de la logique LTL pour la vérification, la logique CTL* est en pratique beaucoup moins utilisée.

Les logiques CTL et CTL* correspondent à des logiques du second ordre sur les arbres [134, 110], et en conséquence, les procédures de décisions pour CTL et CTL* utilisent des automates sur les arbres infinis [79], qui ont été initialement développées pour la logique propositionnelle dynamique PDL et le μ-calcul [141].

Comme la logique LTL, la logique propositionnelle dynamique PDL a été introduite afin de raisonner sur les programmes [107, 52]. L'intérêt de la logique PDL n'est pas tant l'utilisation qui en est faite, mais les concepts qui seront réutilisés par la suite dans de nombreuses logiques. En particulier, la *Fischer-Ladner closure* et la *small-model property* de [52], qui sont à la base des techniques de preuves sur la décidabilité de PDL, sont réutilisées pour pratiquement toutes les logiques temporelles [44]. La logique PDL introduit une notion de relation de transition assez générale, qui définit le changement d'états via l'exécution de programmes. La variante la plus

étudiée est basée sur des programmes formés par des expressions régulières sur un alphabet d'instructions basiques [52].

Les logiques ATL et ATL* [5] sont une généralisation des logiques CTL et CTL* qui introduisent une quantification plus fine basée sur une abstraction de la logique PDL. Au lieu de considérer que les transitions sont effectuées par des programmes dont la sémantique est connue, seuls les causes et les effets sont supposés connus, et les programmes sont représentés par des symboles abstraits. Par exemple, la formule ATL $A[a]G\varphi \wedge E[b]\neg\varphi$ signifie que la formule φ doit être satisfaite sur tous les chemins qui sont le résultat de l'exécution du programme a, est qu'il doit exister une exécution du programme b pour laquelle la formule φ n'est pas satisfaite. La logique ATL permet donc d'unifier la logique CTL* avec la logique PDL. Cette abstractin permet également un rapprochement naturel avec la théorie des jeux à plusieurs joueurs.

Le μ-calcul est un formalisme basé sur l'utilisation de point-fixes [77], dont l'expressivité est très grande car elle englobe CTL*. Le lien avec PDL est direct dans [86], mais les modalités dynamiques sont considérées comme une variantes dans [87]. Le μ-calcul avec modalités dynamiques englobe pratiquement toutes les logiques temporelles connues. En plus de CTL* et consœurs, le lien avec la logique PGL [101] est décrit dans [87], et le lien avec la logique HML [67] est décrit dans [22].

En particulier, les notions de point-fixe introduites pour le μ-calcul ont été utilisées pour augmenter l'expressivité de la logique LTL. La logique νTL [11] est la version linéaire du μ-calcul, qui permet d'étendre l'expressivité aux langages ω-réguliers, et la logique μTL [137] est l'augmentation de la logique νTL avec l'opérateur **S** qui permet de gagner en concision, mais nécessite d'utiliser des automates *2-way*.

La figure 4.6 illustre les relations entre les différentes logiques. Les flèches indiquent comment passer d'une logique à une autre. Par exemple, le passage de LTL à νTL se fait par l'ajout d'opérateurs point-fixe. La même modification de CTL* conduit au μ-calcul.

4.6 Conclusion

Dans ce chapitre, nous avons décrit les logiques classiques et temporelles. Nous tentons en particulier de proposer des notations qui s'appliquent autant à la logique propositionnelle qu'à la logique du second ordre ou à la logique temporelle propositionnelle linéaire. Nous nous concentrons principalement sur la syntaxe et la sémantique des formalismes logiques que nous détaillons, ainsi qu'aux complexités de leurs problèmes de satisfaction et de vérification. En particulier, nous ne traitons pas des axiomatisations et des méthodes de preuve associées aux logiques temporelles.

Il existe de nombreux algorithmes associés aux divers problèmes de chaque logique. Nous évoquons un algorithme pour la logique temporelle linéaire, car cette logique sera la logique de référence dans les contributions. Pour certaines logiques, l'implémentation d'algorithmes de satisfaction est un problème difficile. En particulier, le problème de satisfaction du μ-calcul n'a été complètement traité que récemment [53].

Il est intéressant de noter que malgré la diversités des formalismes existants, leurs expressivités sont bornées par celle des langages ω-réguliers pour ce qui concerne les logiques à temps linéaire, et bornées par l'adaptation de la notion de langages ω-réguliers aux arbres infinis pour ce qui concerne les logiques arborescentes. La classe des langages ω-régulier semble donc être un bon point de repère pour juger de l'expressivité d'un formalisme, et les logiques existantes une source

d'inspiration pour la mise en place de nouvelles méthodes algorithmiques exploitant le paradigme de la programmation par contraintes.

Chapitre 5

Formalismes apparentés

Sommaire

Dans ce chapitre, nous évoquons quelques formalismes en rapport avec la notion de contraintes et de séquences infinies. Il est assez difficile de couvrir l'intégralité des travaux algorithmiques qui ont pu être effectués sur des problèmes à dimension temporelle. Nous pensons que malgré l'hétérogénéité de ceux-ci, il est possible de mettre la plupart d'entre eux en relation avec des concepts plus fondamentaux, et c'est ce que nous nous attacherons à faire ici pour une sélection d'entre eux. Nous pensons qu'à terme, une telle mise en relation permettrait de mieux appréhender l'expressivité et la concision des formalismes, ainsi que leur efficacité algorithmique.

La section 5.1 discute de la planification. Les considérations évoquées en planification se généralisent aux CSPs à horizon bornés qui sont traités dans la section 5.2. Certaines généralisations des logiques temporelles sont traitées dans la section 5.3. La notion d'optimisation est traitée dans la section 5.4, et un formalisme équationnel est détaillé dans la section 5.5. Enfin, des considérations relatives à la notion de contrôle sont décrite dans la section 5.6.

5.1 Planification

Le problème de planification classique consiste à construire des séquences d'actions dans un système de transitions qui mènent d'un état initial vers un ensembles d'états but [98]. Ce problème est équivalent au problème du calcul d'un chemin dans un graphe dirigé, mais l'objectif de la planification consiste à déterminer des formalismes et des algorithmes qui permettent de gérer efficacement de grands espaces d'états.

Le formalisme classique consiste à décrire l'ensemble des états de façon implicite par des ensembles de variables propositionnelles, et les actions par des préconditions et des effets immédiats (ajout/suppression de propositions). Cette représentation permet de naviguer l'espace de recherche à partir d'un état initial ou en partant des états but sans avoir à calculer explicitement l'espace d'état. Comme le nombre d'actions faisables dans chaque état est généralement relativement limité, la taille de l'espace à explorer reste raisonnable pour de petits problèmes. Une observation clé est que la plupart des actions ont des effets très limités sur les états, ce qui rend l'examen de l'union de leurs effets utiles pour la recherche. Les états successeurs représentent alors un sur-ensemble des possibilités futures, et la restriction des ensembles d'actions antérieures permet alors de modifier l'ensemble des possibilités afin de satisfaire une des conditions but. Cette observation est à la base de l'algorithme utilisé par GraphPlan [19].

Le formalisme de la planification classique suppose un système déterministe avec état initial et ensemble d'états but, ce qui est convenable pour étudier les systèmes de représentations, mais est très limitant d'un point de vue applicatif. Les problèmes de planification usuels concernent des systèmes nondéterministes avec des conditions d'objectifs plus générales, et des contraintes temporelles sur l'exécution du système. Cependant, les types de contraintes temporelles utilisées en planification sont différentes de celles utilisées en *model checking*.

L'algèbre des points temporels (*point algebra*) [148] permet d'établir des contraintes sur des instants. Les relations primitives sont les relations linéaires $P = \{<, =, >\}$, et les relations générales sont constituées de tous les sous-ensembles de P. Étant donné un ensemble d'instants $\{t_1, \ldots, t_n\}$ et un ensemble de contraintes $t_i \lesseqgtr_{ij} t_j$ avec $\lesseqgtr_{ij} \in 2^P$ pour $1 \leqslant i, j \leqslant n$ et $t_i \lesseqgtr_i c_i$ avec $\lesseqgtr_i \in 2^P$ et $c_i \in \mathbb{N}$ pour $1 \leqslant i \leqslant n$, une table de composition permet de dériver les contraintes pour toutes les paires d'instants.

Exemple 5. Pour donner une sémantique de planification à un tel ensemble de contraintes, il suffit de considérer que les instants représentent des débuts ou des fins de tâches. Par exemple, soient T_1, T_2 et T_3 trois tâches, avec les contraintes que T_2 doit être effectuée après T_1, mais que T_3 peut commencer après le début de T_1, mais doit finir avant que T_2 ne commence. Alors, en représentant par $b(T_i)$ et $e(T_i)$ les instants de début et de fin d'une tâche T_i, il suffit de modéliser le problème par les contraintes suivantes :

$$\begin{gathered} b(T_2) < e(T_1) \\ b(T_3) \geqslant b(T_1) \\ e(T_3) < b(T_2) \end{gathered}$$

L'algèbre d'intervalles de Allen [3] permet de raisonner au niveau de la notion d'intervalles. Cette algèbre comporte 13 relations de base qui correspondent à toutes les possibilités cohérentes de deux intervales. Comme une relation générale est une disjonction de ces relations, la table de composition comporte 8192 entrées, ce qui rend préférable l'utilisation d'un système de raisonnement même en ce qui concerne la composition des relations.

Exemple 6. En considérant les tâches T_i, $i \in \{1, 2, 3\}$ de l'exemple 5 comme des intervalles $[b(T_i), e(T_i)]$, les contraintes d'intervalles sur ces tâches sont les suivantes :

$$\begin{gathered} \texttt{before}(T_1, T_2) \\ \texttt{overlap}(T_1, T_3) \\ \texttt{before}(T_3, T_2) \end{gathered}$$

Un autre formalisme plus général est celui des réseaux de contraintes temporelles [36] dans lequel une contrainte temporelle est une disjonction de contraintes basiques $t[a,b]u \equiv_{\text{def}} a \leqslant u - t \leqslant b$ qui peuvent être combinées avec les opérateurs de composition $(t[a,b]u) \bullet (u[c,d]v) = t[a+c, b+d]v$ et $(t[a,b]u) \cap (t[a',b']u) = t[\max\{a,a'\}, \min\{b,b'\}]u$. Dans ce formalisme, on peut par exemple imposer que la tâche T_2 de l'exemple 5 commence au moins deux unités de temps après le début de T_1 en utilisant la contrainte $b(T_1)[2, t_{\max}]b(T_2)$ où $t_{\max}$ représente une borne supérieure sur l'horizon du problème.

L'algèbre des points et l'algèbre d'intervalles sont des systèmes de contraintes qualitatives, car ils ne permettent pas d'inclure des informations quantitatives dans les contraintes, contrairement aux réseaux de contraintes temporelles. L'algèbre d'intervalles est utilisée dans le langage de planification PDDL [59] ainsi que dans les problèmes d'ordonnancement [76]. Comme la logique temporelle propositionnelle LTL est elle aussi qualitative, il est naturel d'examiner sa relation avec l'algèbre d'intervalles. Le problème est que la notion d'intervalle est plus structurée que la notion de proposition instantanée, et contient implicitement une notion de continuité dans le temps qu'il est difficile d'exprimer en logique LTL. L'inclusion de l'algèbre d'intervalle dans LTL décrite dans [113] (2006) utilise des variables propositionnelles additionnelles pour chaque intervalle.

Exemple 7. Afin d'exprimer le problème 5 en logique LTL, nous utilisons les trois variables propositionnelles p_1, p_2 et p_3, et l'ensemble de contraintes $\{\neg p_i \mathbf{U}(p_i \mathbf{U}(\neg p_i)) \mid i \in \{1,2,3\}\}$ pour induire la sémantique d'intervalles sur p_1, p_2 et p_3. Les expressions LTL des contraintes sur les intervalles sont alors les suivantes :

$$\begin{aligned} \texttt{before}(T_1, T_2) &\equiv \mathbf{F}(p_1 \wedge \mathbf{F}(\neg p_1 \wedge \neg p_2 \wedge \mathbf{F} p_2)) \\ \texttt{before}(T_3, T_2) &\equiv \mathbf{F}(p_3 \wedge \mathbf{F}(\neg p_3 \wedge \neg p_2 \wedge \mathbf{F} p_2)) \\ \texttt{overlap}(T_1, T_3) &\equiv \mathbf{F}(p_1 \wedge \neg p_3 \wedge \mathbf{F}(p_1 \wedge \neg p_3 \wedge \mathbf{F}(\neg p_1 \wedge p_3))) \end{aligned}$$

Un autre problème consiste à intégrer la logique LTL en planification [100] (2011). Il est assez surprenant que, étant donnés la notoriété des problèmes de planification, des problèmes de *model checking*, et le rapport évident que les deux entretiennent avec les notions de systèmes de transitions et de propriétés temporelles, un tel lien n'ai pas été effectué plus tôt, d'autant plus que l'utilisation des techniques de *model checking* pour la résolution de problèmes de planification est déjà évoquée dans [98] (2004).

5.2 Programmation par contraintes sur horizon borné

La résolution de problèmes de planification par les techniques de satisfaction propositionnelle ou de programmation par contraintes est connue [98, 13]. Le principe consiste à encoder le problème sur un horizon fini puis utiliser un solveur pour rechercher un chemin satisfaisant les contraintes de transition et menant à un état but. Le procédé est réitéré sur des horizons de plus en plus grands jusqu'à ce qu'une solution soit trouvée ou que la taille du problème dépasse les capacités du solveur. Cette approche permet d'inclure facilement des contraintes temporelles diverses ainsi que des contraintes d'optimisation, tout en s'améliorant naturellement via l'amélioration des techniques de satisfaction et de programmation par contraintes.

Un problème comportant des contraintes temporelles est généralement représentable par un système de transitions dont les séquences d'états doivent satisfaire des conditions spécifiées. Un

état correspond à un ensemble de variables, qui sont dupliquées pour chaque instant considéré dans le problème. La longueur de la séquence est l'horizon du problème. Nous distinguons les horizons finis/bornés, les horizons non bornés, et les horizons infinis.

Dans le cas d'un horizon fini ou borné, l'approche de résolution par CSP est complète. Un exemple de CSP à horizon fini est celui qui permet la génération d'un emploi du temps. Les problèmes d'ordonnancement avec capacités sont généralement des problèmes d'optimisation dans lesquels la borne supérieure de l'horizon est la somme de la durée maximale de chaque tâche, et pour lesquels l'objectif consiste à minimiser la durée nécessaire à la réalisation de toutes les tâches. Les problèmes de planification sont des problèmes à horizon non borné, dans le sens où la longueur maximale de la séquence est tellement grande qu'il est équivalent de considérer qu'elle est finie mais inconnue. En conséquence, l'absence de solution sur un horizon donné n'implique pas que le problème n'admet pas de solution.

Il est également possible de considérer le cas des horizons infinis. L'approche employée en programmation par contraintes consiste alors généralement à rechercher des solutions ultimement périodiques, de la forme uv^ω, où u et v sont des mots finis, ce qui permet de ramener les problèmes aux cas précédents. Il suffit pour cela d'augmenter la recherche de chemins finis d'une contrainte de continuité qui relie le dernier état à un état déjà visité. L'inconvénient d'une telle approche est qu'elle empêche la production de solutions apériodiques, ce qui est n'est pas techniquement limitant car toute machine à mémoire finie qui admet un fonctionnement infini admet un fonctionnement ultimement périodique [133]. Cependant, ce procédé limite fortement le type des solutions produites. Les approches pour la résolution de problèmes d'ordonnancement cycliques sont basés sur ces hypothèses [20].

5.3 Logiques temporelles concrètes

La logique temporelle linéaire propositionnelle LTL est une logique temporelle sur le domaine $\mathbb{B}$ des booléens, qu'il est naturel de généraliser à d'autres domaines, finis ou infinis.

Comme pour la logique du premier ordre, la généralisation à des domaines arbitraires implique de distinguer les valeurs des variables des valeurs de vérités des formules logiques, et ceci se fait en introduisant une notion de terme et de prédicat. À la différence de la logique du premier ordre, l'élément le plus naturel est l'intoduction de références temporelles relatives afin de relier les valeurs des variables temporelles à des instants différents.

Syntaxe des logiques LTL sur des domaines concrets. Soit $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ une structure et $\mathcal{V}$ un ensemble dénombrable de variables. La syntaxe de la logique LTL sur les domaines concrets est la suivante :

- $v \in \mathcal{V}$ est une **variable** ;
- Si v est une variable, $\mathbf{X}v$ est une **variable** ;
- Soit $P \in \mathcal{P}$ un prédicat d'arité $\mathrm{ar}(P) = k$ et $\langle v_1, \ldots, v_k \rangle \in \mathcal{V}^k$ un tuple de k variables. Alors $\mathcal{P}(v_1, \ldots, v_k)$ est un **terme** ;
- Un terme est une **formule** ;
- Si φ est une formule, alors $\mathbf{X}\varphi$ est une **formule** ;
- Si φ et ψ sont des formules, alors $\varphi \barwedge \psi$ et $\varphi \mathbf{U} \psi$ sont des **formules**.

Sémantique des logiques LTL sur les domaines concrets La sémantique des logiques LTL sur les domaines concrets est similaire à celle de la logique LTL sur les variables propositionnelles. Une interprétation d'un ensemble de variables X est une fonction $I : X \to \mathcal{D}$, et une interprétation temporelle de X est une fonction $I_T : \mathbb{N} \to \mathcal{I}(X)$ où $\mathcal{I}(X)$ représente l'ensemble des interprétations des variables. Si $T = P(\mathbf{X}^{t_1} v_1, \ldots, \mathbf{X}^{t_k} v_k)$ est un terme et I est une interprétation temporelle sur $\{v_1, \ldots, v_k\}$, I satisfait T si et seulement si :

$$\langle I(t_1)(v_1), \ldots, I(t_k)(v_k)\rangle \in P$$

Comme c'est le cas pour les logiques du premier ordre, les logiques temporelles sur des domaines concrets concernent surtout les domaines infinis, et en premier lieu, les entiers naturels, les entiers relatifs, les réels, et divers types de prédicats parmi lesquels la relation d'ordre linéaire, la fonction successeur et l'addition.

Le problème de satisfaction de la logique LTL sur $\langle \mathbb{R}, \{<, =\}\rangle$ est PSPACE-complet, et cela est vrai pour toutes les structures pour lesquelles toute solution partielle d'un ensemble de contraintes consistant peut être étendue à une solution [10].

Cette propriété n'est pas satisfaite par les structures $\langle \mathbb{N}, \{<, =\}\rangle$ et $\langle \mathbb{Z}, \{<, =\}\rangle$, mais le problème de satisfaction est aussi PSPACE-complet pour ces structures [39]. Les mêmes auteurs montrent que le problème de satisfaction devient indécidable pour les structures $\langle \mathbb{N}, \{<, =, <_{+1}\}\rangle$ et $\langle \mathbb{Z}, \{<, =, <_{+1}\}\rangle$, où '$<_{+1}$' est une relation qui généralise la relation successeur $S(x, y) \equiv_{\text{def}} y = x + 1$ et qui représente un système de comptage arbitraire.

L'arithmétique de Presburger $\langle \mathbb{N}, \{<, =, +\}\rangle$ devient aussi indécidable dans le contexte temporel, car elle permet de simuler les machines de Minsky, un modèle de calcul Turing-complet. Les restriction qu'il est possible d'appliquer sont extrêment variées, et conduisent à une grande variété de formalismes dont l'étude est relativement récente [38].

En introduisant la quantification, les points-fixes ou les modalités dynamiques, des problèmes similaires peuvent être étudiés dans le contexte des autres logiques temporelles qui ont été décrites dans le chapitre 4.

En ce qui concerne les domaines finis, [7] présente un formalisme de simulation qualitative utilisant la programmation par contraintes. L'originalité de l'approche réside dans la modélisation, mais la méthode de résolution est classique, car le problème est résolu par encodage temporel sur un horizon borné, avec recherche de solutions ultimement périodiques.

À l'opposé, il existe des travaux utilisant la programmation par contraintes, mais pour manipuler des domaines infinis. Dans [37], les auteurs utilisent la programmation logique pour effectuer la vérification CTL de systèmes comportant un nombre infini d'états. Le système est modélisé par un programme logique, et la sémantique de la logique CTL est implémentée par des transformation de programmes logiques dont les point-fixe représentent l'ensemble des états satisfaisant la spécification. Le cas de la logique LTL correspond à la restriction des modalités CTL aux opérateurs **AF** et **AG**, mais les caractérisations point-fixe sont données seulement pour les modalités **EF** et **EG** et les autres sont calculées par complémentation, ce qui rend l'approche décrite peu convaincante pour la logique LTL sans un travail supplémentaire. En outre, le formalisme décrit repose entièrement sur les clauses de Horn pour décrire les transitions entre états, ce qui est difficile à utiliser dans le contexte de la programmation par contraintes pure.

$$
\begin{array}{ll}
(1) & f(\sigma+\tau) = f(\sigma) + f(\tau) \\
(2) & n(\sigma+\tau) = n(\sigma) + n(\tau) \\
(3) & f(\sigma \otimes \tau) = f(\sigma) \times f(\tau) \\
(4) & n(\sigma \otimes \tau) = (n(\sigma) \otimes \tau) + (\sigma \otimes n(\tau))
\end{array}
$$

FIGURE 5.1 – Addition, produit de convolution

5.4 Optimisation

La notion de contrainte sur les séquences infinies implique naturellement la notion d'optimisation. À la différence de l'optimisation sous contraintes de la PPC, tous les critères d'optimisation ne sont pas utilisables, car le nombre de séquences infinies sur un ensemble non trivial est indénombrable et il est possible de définir des critères d'optimisation qui n'admettent aucun optimum.

Exemple 8. Soit $\mathcal{S} = \langle \mathcal{D}, \{\neq\} \rangle$ avec $\mathcal{D} = \{0,1,2,3,4,5,6,7,8,9\}$. Soit $\varphi = \mathbf{F}(v \neq 0)$ une formule temporelle sur $\mathcal{S}$, et f la fonction qui associe à toute interprétation temporelle $\sigma = \sigma[0]\sigma[1]\ldots$ de $\text{var}(\varphi) = \{v\}$ le nombre réel $f(\sigma) = \sum_{i>0} \sigma[i](v) \times 10^{-i}$. Le problème consiste à miniser f en satisfaisant φ, ce qui revient à trouver le nombre réel différent de 0 le plus petit possible. Comme pour tout $r > 0$, il existe $r' > 0$ tel que $r > r'$, l'optimum de cette fonction n'existe pas.

L'exemple 8 montre la nécessité de définir des classes de critères d'optimisation pour lesquels les optimums existent. Dans [28], les auteurs étudient la synthèse de stratégies finies dans le cadre de jeux sur des graphes dont les transitions sont étiquetées par des tuples d'entiers relatifs représentant des variations d'énergies, et dont l'objectif consiste à conserver les sommes ou les moyennes au dessus de seuils donnés. La méthode de résolution consiste à déterminer des points de retour dans le développment infini du graphe telle que tous les cycles qui en résultatent garantissent que les conditions sont satisfaites quelques soient les actions du joueur universel. Dans un contexte de satisfaction de contraintes, le problème est simplifié de par l'absence du joueur universel, mais la méthode algorithmique reste essentiellement la même, car le développement de l'arbre de transition est nécessaire. Cette approche montre qu'il est possible de calculer des séquences infinies pour lesquelles les sommes et moyennes de certains indicateurs sont contraintes. Cependant, ceci ne permet pas d'établir qu'il est possible d'utiliser de tels indicateurs dans un but d'optimisation.

5.5 Equations sur les séquences infinies

Il existe aussi des travaux qui s'intéressent aux séquences infinies d'un point de vue équationnel. Dans [116], une séquence infinie (*stream*) $\sigma \in \Sigma^\omega$ est considérée comme un objet dont il est possible d'accéder seulement à la valeur initiale $f(\sigma) = \sigma[1]$ [2] et au reste de la séquence $n(\sigma) = \sigma[2,\omega]$. Diverses opérations sur les séquences sont définies par des *équation différentielle comportementales*, c'est-à-dire des systèmes de deux équations dont l'une spécifie la condition initiale et l'autre spécifie comment construire la valeur suivante. À titre d'exemple, nous donnons les équations pour l'addition ((1) et (2)) et le produit de convolution ((3) et (4))dans la figure 5.1.

2. Index modifié pour être conforme aux notations introduites dans le chapitre 2

$$
\begin{aligned}
f(n^2(\pi)) &= f(n(\underbrace{n(\sigma \otimes \tau)}_{(4)})) \\
&= f(\underbrace{n((n(\sigma) \otimes \tau) + (\sigma \otimes n(\tau)))}_{(2)}) \\
&= \underbrace{f(n(n(\sigma) \otimes \tau) + n(\sigma \otimes n(\tau)))}_{(1)} \\
&= f(\underbrace{n(n(\sigma) \otimes \tau)}_{(4)}) + f(\underbrace{n(\sigma \otimes n(\tau))}_{(4)}) \\
&= \underbrace{f((n(n(\sigma)) \otimes \tau) + (n(\sigma) \otimes n(\tau)))}_{(1)} \\
&+ \underbrace{f((n(\sigma) \otimes n(\tau)) + (\sigma \otimes n(n(\tau))))}_{(1)} \\
&= \underbrace{f(n(n(\sigma)) \otimes \tau)}_{(3)} + \underbrace{f(n(\sigma) \otimes n(\tau))}_{(3)} \\
&+ \underbrace{f(n(\sigma) \otimes n(\tau))}_{(3)} + \underbrace{f(\sigma \otimes n(n(\tau)))}_{(3)} \\
&= f(n(n(\sigma))) \times f(\tau) + f(n(\sigma)) \times f(n(\tau)) \\
&+ f(n(\sigma)) \times f(n(\tau)) + f(\sigma) \times f(n(n(\tau))) \\
&= \textstyle\sum_{0 \leqslant i \leqslant 2} \binom{2}{i} f(n^i(\sigma)) \times f(n^{2-i}(\tau))
\end{aligned}
$$

FIGURE 5.2 – Évaluation du produit de convolution

Exemple 9 (Évaluation du produit de convolution)**.** Soit σ et τ deux séquences infinies et $\pi = \sigma \otimes \tau$. L'objectif consiste à calculer les valeurs $f(n^k(\pi))$ pour des valeurs de k données. Les équations données dans la figure 5.1 permettent d'effectuer ces évaluations par récursion. Nous détaillons le procédé pour $k = 2$ dans la figure 5.2.

Il est possible de montrer que de façons générale,

$$f(n^k(\pi)) = \sum_{0 \leqslant i \leqslant k} \binom{k}{i} f(n^i(\sigma)) \times f(n^{k-i}(\tau))$$

Le formalisme est développé principalement pour les nombres réels. D'une façon générale, les équations différentielles comportementales permettent de définir un flux *unique* dont la k-ème valeur est fonction des k premières valeurs de chaque opérande. Ceci offre la possibilité théorique d'afficher le résultat d'une opération au fur et à mesure que les valeurs des séquences d'entrées sont connues, d'où le terme de "flux" (*stream*). Des applications concernent la résolution d'équations aux différences sur les réels, ou d'équations différentielles en utilisant les expansions de Taylor des fonctions.

Une correspondance particulière avec des système de transitions dont les états et transitions sont étiquetés par des réels permet d'établir une correspondance graphique entre des séquences dites "rationnelles" et les états d'un automate. La correspondance effectuée consiste à utiliser la valeur associée à un état comme valeur initiale et les transitions pour définir les valeurs suivantes en utilisant les séquences associés aux autres états pondérées par les valeurs des transitions.

Ce formalisme est un système de spécification de séquence infinies sur les réels. Cependant, les solutions des équations sont par construction unique, ce qui le rend particulier du point de vue de la satisfaction de contraintes, et similaire dans l'idée au constructions utilisées dans le langage Haskell [72] ou dans des langages synchrones tels que Lustre [66].

Ce formalisme équationel est généralisé à un formalisme multivariable qui permet de décrire des langages valuées des alphabet finis, c'est-à-dire des langages dotés d'une fonction $v : \Sigma* \to \mathbb{R}$ qui associe à tout mot fini un poids. Les séquences infinies considérées sont équivalentes à des séries formelles, de la forme $\sigma = \sum_{w \in \Sigma^*} v(w) \cdot w$, où w représente une variable formelle et $v(w)$ représente sa multiplicité.

5.6 Synthèse de contrôleur

La problématique principale de la théorie du contrôle discret est la conception de contrôleur pour des sytèmes à évènements discrets, pour lesquels le contrôleur doit garantir qu'un système se comporte en respectant des propriétés spécifiées. Le problème est par conséquent très apparenté au problème du *modèle checking*, mais la différence principale est que le problème de décision est ici rempacé par le problème de synthèse d'une restriction d'un système qui satisfait des propriétés données.

Une approche consiste à spécifier séparément le système, le contrôleur et les propriétés qui doivent être satisfaites, puis à vérifier que le système contrôlé satisfait bien les propriétés [112]. Le problème de synthèse de contrôleur consiste à considérer les propriétés à vérifier directement comme une spécification du contrôleur, et de s'en servir pour générer un contrôleur valide par construction. Ceci permet de garantir que les propriétés spécifiées seront vérifiées par construction, ou de détecter que le système donné n'est pas contrôlable. Dans le second cas, il est alors facile de modifier les spécifications ou le système pour obtenir rapidement le système contrôlé correspondant. Comme ils existe plusieurs contrôleurs pour un même système, le problème de la synthèse de contrôleur correspond à un problème d'optimisation si l'on souhaite par exemple minimiser la taille du contrôleur produit.

Dans [93], la programmation par contraintes est utilisée pour résoudre le problème de la synthèse de contrôleur en environnement non déterministe partiellement observable. Un modèle de contrôle est un tuple (S, O, C, I, T, F) dans lequel S est l'ensemble des variables d'états, $O \subset S$ est le sous-ensemble des variable d'états observables, C est l'ensemble des variables de contrôle, I est une relation d'initilisation unaire, et T et F sont les relations binaires de transition et de faisabilité. Cette modélisation se transcrit directement sous la forme d'un CSP, dans lequel les variables sont S et C, et les contraintes sont spécifiées par les relations I, T et F. L'objectif est alors de trouver une politique Π qui associe à chaque observation de O une commande dans C de façons à ce que F soit satisfaite.

Cette formulation permet de calculer des politiques sans mémoire, ce qui a un intérêt limité lorsque l'environnement est partiellement observable. Une manière d'ajouter de la mémoire au contrôleur consiste à ajouter une variable q de domaine $[1 \dots N]$. L'ajout de mémoire engendre une augmentation rapide de la taille de l'espace de recherche, et il est donc préférable d'augmenter progressivement la borne maximale sur la taille des contrôleurs cherchés.

Dans [93], les auteurs considèrent des problèmes de contrôles avec propriétés de sûreté, propriétés orientés but, et conjonction de ces deux types de propriétés. Dans les deux derniers cas, les auteurs vérifient explicitement que le système contrôlé ne peut pas engendrer de trajectoires

infinies. L'algorithme de synthèse de contrôleur, dit "brancher et simuler" consiste à générer des politiques partielles, puis simuler le système contrôlé jusqu'à ce que des états non couverts par la politique soient rencontrés, et répéter l'opération jusqu'à ce que tous les états rencontrés soient contrôlés. Dans le cas des problèmes orientés but, la présence de boucles lors de la simulation signale des politiques incorrectes. Dans le cas des problèmes orientés sûretés, la présence d'état ne satisfaisant pas les propriétés spécifiées signale des politiques incorrectes.

L'idée de chercher à synthétiser des contrôleurs avec quantité de mémoire bornée provient de deux observations. D'une part, les contrôleur sans mémoire sont d'un intérêt limité. D'autre part, considérer l'historique global des actions, tels que cela est fait dans les QCSP [106], conduit en général à une politique d'une taille beaucoup trop grande, voire infinie dans le contexte des problèmes orientés sûreté. La notion intermédiaire d'état de croyance conduit à des contrôleur à mémoire finie, mais la borne supérieure utilisée est beaucoup trop grande. L'utilisation d'une petite borne supérieure donnée à l'avance permet de faire abstraction de la sémantique associée à l'état du contrôleur, qui dans ce contexte n'a pas beaucoup d'importance.

Une autre approche d'une complexité plus simple est décrite dans [94]. La difficulté intrinsèque du problème de contrôle est la relation d'atteignabilité. Un contrôleur valide est un contrôleur qui garantit les bonnes propriétés seulement dans les états atteignables. Un contrôleur simplement valide est un contrôleur qui garantit les bonnes propriétés dans tous les états, y compris ceux qui ne sont pas atteignables. Il est immédiat qu'un contrôleur simplement valide est nécessairement valide, et qu'il peut exister un contrôleur valide lorsqu'il n'existe pas de contrôleur simplement valide. Il en résulte que l'approche utilisée dans [94] est plus simple à mettre en oeuvre, mais moins générale que celle utilisée dans [93].

Exemple. Nous présentons le problème de la synthèse de contrôleur sur un exemple décrit dans [93], afin de comparer l'approche utilisée dans ces travaux avec la nôtre.

Il s'agit d'un problème de synthèse de contrôleur pour un robot qui peut se déplacer dans une grille rectangulaire de dimensions 3×2 dotée de murs. Le robot doit atteindre une case particulière de la grille en évitant de passer par une certaine case, mais est dans l'impossibilité de savoir précisément la case dans laquelle il se trouve. Les seules informations qui sont à sa disposition sont les informations relatives à la présence de murs. Les états du système sont décrits par les variables $S = \{x, y, w_N, w_S, w_E, w_W, qs\}$, où :

— x et y représentent les coordonnées du robot dans la grille
$\mathbf{d}(x) = [1 \dots 3]$
$\mathbf{d}(y) = [1 \dots 2]$

— w_N, w_S, w_E et w_W représentent la présence de mur au nord, au sud, à l'est et à l'ouest respectivement
$\mathbf{d}(w_N) = \mathbf{d}(w_S) = \mathbf{d}(w_E) = \mathbf{d}(w_W) = \{0, 1\}$

— qs représente l'état interne du contrôleur
$\mathbf{d}(qs) = [1 \dots N]$

Parmi les variables de S, seules les variables w_N, w_S, w_E, w_W et qs sont observables. Les coordonnées du robots x et y doivent être "déduites" par le contrôleur en fonction des observations. Ceci n'implique pas que les déductions du contrôleurs doivent être exactes, il est seulement important que les exigences imposées au contrôleur soient satisfaites.

L'ensemble des variables de contrôle est $C = \{m, qc\}$, où m une variable de domaine $\mathbf{d}(m) = \{m_N, m_S, m_E, m_W\}$ qui ordonne au robot de se déplacer au nord, au sud, à l'est, ou à l'ouest,

$$T = \bigwedge \begin{cases} x' = x + (m = m_E) - (m = m_W) \\ y' = y + (m = m_N) - (m = m_S) \\ w'_N \leftrightarrow (y' = 2 \vee (y' = 1 \wedge x' = 2)) \\ w'_E \leftrightarrow (x' = 1) \\ w'_S \leftrightarrow (y' = 1 \vee (y' = 2 \wedge x' = 2)) \\ w'_W \leftrightarrow (x' = 3) \\ qs' = qc \end{cases}$$

FIGURE 5.3 – Relation de transition T

$$F = \bigwedge \begin{cases} w_N \rightarrow (m \neq m_N) \\ w_S \rightarrow (m \neg m_S) \\ w_E \rightarrow (m \neq m_E) \\ w_W \rightarrow (m \neq m_W) \end{cases}$$

FIGURE 5.4 – Relation de faisabilité F

et qc est une variable de domaine $\mathbf{d}(qc) = [1 \dots N]$, qui spécifie le prochain état interne du contrôleur.

La relation T (Figure 5.3) décrit l'évolution du système, et contient une description de la topologie de la grille. Les coordonnées x et y sont une fonction des coordonnées courantes et de l'action m effectuée par le robot. La présence de murs dépend de la position courante du robot, et l'état interne du contrôleur qs est directement spécifié par la variable de contrôle qc.

La relation de faisabilité F (5.4) décrit les actions possibles, en l'occurence simplement qu'il est impossible de se déplacer dans une direction où il y a un mur.

Enfin, la relation d'initialisation I (5.5) décrit les états initiaux du systèmes. Dans cet exemple, le robot commence dans les positions $(2, 1)$ ou $(2, 2)$, il y a des murs au nord et au sud, mais ni à l'est, ni à l'ouest, et l'état interne du contrôleur vaut initialement 1.

Les relation T, F et I spécifie le système non contrôlé, c'est-à-dire le "modèle de contrôle". Les exigences sont spécifiées par les relations G et R (Figure 5.6), où G spécifie les exigences de but, ici que le robot doit atteindre la case $(2, 2)$, et R spécifie les exigences de sûreté, ici que le robot doit éviter la case $(3, 1)$.

Exécution de l'algorithme Le principe de l'algorithme consiste à synthétiser des politiques partielles et simuler le système à la recherche d'états non couverts par la politique.

L'exécution commence sur une politique vide. La simulation consiste à parcourir le système à partir des états initiaux, ce qui implique de calculer l'ensemble des états initiaux. Dans le cas présent, les états initiaux sont les états $s_1 = \{x = 2, y = 2, w_N, w_S, \neg w_E, \neg w_W, qs = 1\}$ et

$$I = \bigwedge \begin{cases} x = 2 \\ w_N \wedge w_S \wedge \neg w_E \wedge \neg w_W \\ qs = 1 \end{cases}$$

FIGURE 5.5 – Relation d'initialisation I

$$G = (x = 2 \wedge y = 2)$$
$$R = \neg(x = 3 \wedge y = 2)$$

FIGURE 5.6 – Exigences de but G et exigences de sûreté R

$$\Pi(w_N, w_S, \neg w_E, \neg w_W, qs = 1) = \{m = m_W, qc = 1\}$$
$$\Pi(w_N, \neg w_S, \neg w_E, w_W, qs = 1) = \{m = m_E, qc = 2\}$$
$$\Pi(w_N, w_S, \neg w_E, \neg w_W, qs = 2) = \varepsilon$$
$$\Pi(\neg w_N, w_S, \neg w_E, w_W, qs = 1) = \{m = m_N, qc = 1\}$$

FIGURE 5.7 – Politique valide pour l'exemple du robot

$s_2 = \{x = 2, y = 1, w_N, w_S, \neg w_E, \neg w_W, qs = 1\}$, qui ne sont pas couverts par la politique puisque celle-ci est vide. Il est donc nécessaire de définir une action pour le contrôleur correspondant à cet état. Comme le contrôleur ne prend en compte que les variables observables, il faut définir une action pour l'observation $o_1 = \{w_N, w_S, \neg w_E, \neg w_W, qs = 1\}$, qui s'applique à la fois aux états s_1 et s_2.

Une première action consiste à ne rien faire et à déclarer le contrôle comme terminé, c'est-à-dire définir l'action $\Pi(o_1) = \varepsilon$. L'action de ne rien faire implique la vérification des exigences de but, qui ne sont pas satisfaites pour l'état s_1. Ceci implique que la décision de rien faire sur o_1 est incorrecte et doit être révisée.

Les actions possibles du contrôleur sont décrites par l'ensemble $A_P = M \times [1 \dots N]$ où $M = \{m_E, m_W, m_S, m_N\}$, qui est le domaine des variables $\{m, qc\}$. Concernant qc, il est préférable de le laisser à sa valeur minimale, puis d'incrémenter sa valeur au besoin. Les actions faisables sont décrites par l'ensemble $A_F = A_P \cap F = \{m_E, m_W\} \times [1 \dots N]$. Les deux premiers choix à examiner pour l'action à réaliser sur l'observation o sont donc $C_1 = \{m = m_E, qc = 1\}$ et $c_2 = \{m = m_W, qc = 1\}$.

La simulation du système avec l'action $\Pi(o_1) = C_1$ produit l'état $s_3 = \{x = 3, y = 2, w_N, w_E, \neg w_S, \neg w_W, qs = 1\}$. L'action d'aller à l'ouest ($m = m_W$) sans modifier qs engendre une boucle infinie lors de la simulation du système, ce qui implique l'état but ne peut être atteint. L'action d'aller à l'ouest en modifiant qs n'est pas faisable non plus, car elle peut envoyer le robot dans la position interdite. Il est nécessaire de réviser le choix effectuer pour o_1 en examinant $\Pi(o_1) = C_2$.

Par itération, cette procédure produit la politique valide donnée dans la figure 5.7.

Cet algorithme à l'avantage d'être complet et produire une politique de taille minimale. L'inconvénient est l'aspect simulation qui contient le problème du calcul de la clôture transitive d'un graphe dirigé. Ce problème n'est traitable en PPC que par la résolution de plusieurs CSPs. La relation d'initialisation permet de déterminer les états initiaux, et la relation de transition permet de calculer l'ensemble des successeurs. Pour calculer la clôture transitive, il est nécessaire d'ajouter l'ensemble des successeurs trouvés à l'ensemble des états de départ, et de résoudre à nouveau jusqu'à ce qu'aucun nouveau sommet ne soit rencontré. Lorsque le nombre de noeuds du graphe est élevé, comme c'est le cas dans le contexte de la simulation de systèmes, cette méthode est inapplicable.

Un manière plus économe de synthétiser un contrôleur est décrite dans [147]. La méthode évite le problème du calcul de la clotûre transitive en faisant l'hypothèse que tous les états du système sont atteignables. Avec cette hypothèse, le contrôleur produit des politiques "simplement

valides" [94], et le problème de synthèse est dans ce cas amenable à un traitement complet par la PPC, mais nécessite de recourir à la méthode plus générale si la synthèse échoue, car la notion de politique simplement valide est moins générale que celle de politique valide. Les auteurs arguent qu'il est fréquent qu'une politique simplement valide existe lorsqu'une politique valide existe, et que les cas pour lesquels il n'existe aucune politiquement simplement valide mais une politique valide nécessitent une révision de la spécification du problème.

5.7 Conclusion

Nous avons décrit un certains nombre de formalismes à dimension temporelle, et nous pourrions en décrire d'autres. En particulier, nous n'avons pas discuté des formalismes probabilistes [120, 146], des formalismes à temps réel et/ou pondérés [21], ou des formalismes hybrides [68]. Ces axes fournissent d'autres directions pour explorer relations que les divers formalismes entretiennent entre eux, qui sont en grande partie orthogonales à celles qui ont été explorées dans ces deux derniers chapitres. L'ajout de ces qualificatifs à pour principale effet, en général, d'augmenter la complexité des problèmes associer et de leur étude.

Parmi les formalismes que nous avons évoqués, les liens que ceux-ci entretiennent avec la programmation par contraintes sont assez divers, mais la tendance générale que celle-ci permet de résoudre plus efficacement des problèmes plus limités. En particulier, ceci est le cas pour les problèmes d'ordonnancement, de planification, et de synthèse de contrôleur. Nous verrons que nous sommes pour le moment dans une situation similaire en ce qui concerne la programmation par contraintes sur les variables flux, comparée à l'expressivité qu'il est possible d'atteindre avec les logiques temporelles. Nous verrons aussi que notre formalisme permet de gagner beaucoup en terme de concision.

Deuxième partie

Contributions

Chapitre 6

StCSPs

Sommaire

Dans ce chapitre, nous proposons le formalisme des **StCSP**s (abbrévation de l'anglais *Stream Constraint Satisfaction Problem*) pour modéliser et résoudre des problèmes de satisfaction de contraintes sur des variables flux. Notre formalisme permet de réutiliser les techniques de propagation de contraintes qui sont décrites dans le chapitre 3, ce qui le rend facilement intégrable à des solveur de contraintes existants.

Le travail exposé ici a fait l'objet d'une publication [42]. Un travail similaire avec une approche plus applicative est décrit dans [126], [80] et [84]. Comme nous travaillons dans un cadre plus théorique, nous avons volontairement choisi de limiter la richesse syntaxique afin de permettre une comparaison plus directe avec d'autres formalismes.

Dans ce qui suit, le terme "PPC" désigne la programmation par contraintes usuelle. Les variables PPC et les contraintes PPC désignent les notions de variable et de contrainte au sens de la PPC (cf. Chapitre 3), et sans ce qualificatif, ces notions seront à interpréter dans le contexte des variables flux.

6.1 Hypothèses de travail

Nous souhaitons modéliser et résoudre des problèmes sur des variables flux en utilisant les techniques de la programmation par contraintes. Une valeur d'une variable flux est une séquence infinie de valeurs, qu'il est commode de considérer comme supportée par une structure temporelle.

Les formalismes temporels distinguent le cas d'un temps discret de celui d'un temps continu. Dans une structure à temps discret, la valeur d'une séquence à un instant t est définie seulement pour $t \in \mathbb{N}$. Dans une structure à temps continu, les valeurs des variables sont définies pour tout instant $t \in \mathbb{R}$. Du fait de problèmes de représentation, les formalismes combinatoires à temps continu considèrent généralement des séquences infinies de *changement d'état*, annotées

de contraintes temporelles [4]. Une solution reste alors une séquence infinie de changements d'état indexée par les entiers naturels, et il semble donc naturel de supposer d'abord une structure de temps discret dans le cadre de notre travail.

Nous supposerons également une structure de temps linéaire. Dans le cas d'un problème de satisfaction de contraintes, une structure de temps à branchement supposerait que les solutions du problèmes de satisfaction soient des langages, c'est-à-dire que les ensembles de solutions soient des ensembles de langages sur les mots infinis, comme c'est le cas pour les logiques temporelles arborescentes du chapitre 4. Du fait des relations entre les logiques exposées dans ce même chapitre, et de la prépondérance des structure de temps linéaires, nous pensons qu'il est préférable d'étudier d'abord un formalisme sur une telle structure avant de considérer des formalismes similaires sur des structures de temps arborescentes.

Enfin, une hypothèse très forte consiste à utiliser seulement les techniques de programmation par contraintes dédiées à la résolution du problème de satisfaction de contraintes, et en particulier d'éviter les problèmes liés à la disjonction de contraintes. La disjonction de contraintes est une des limitations majeures de la programmation par contraintes, et nous avons examiné l'état actuel du traitement de la disjonction dans le chapitre 3. Une hypothèse encore plus forte est de limiter au maximum le nombre de résolutions de CSPs. En effet, le recours à la programmation logique ou à la résolution multiple de CSPs permet de masquer les problèmes liés à la disjonction en examinant tous les cas possibles manuellement. La restriction aux problèmes de satisfaction de contraintes conjonctifs permet de mieux mettre en lumière certains aspects des logiques temporelles, et également d'obtenir un algorithme plus élégant et plus adapté à certaines situations. Nous montrerons que cet algorithme peut facilement être adapté à l'approche plus usuelle qui utilise un nombre non borné de résolutions de CSPs pour la recherche de solutions.

6.2 StCSPs

Dans cette section, nous discutons d'abord des diverses représentations possibles pour les variables flux, puis nous donnons la définition générale d'un StCSP. Nous discutons ensuite de la notion générale de contrainte sur les variables flux, et des problèmes liés à l'optimisation.

Variables flux Une variable flux est une variable dont le domaine est un ensemble de séquences infinies. Nous pouvons considérer une variable flux de deux manières.

Une approche consiste à considérer une variable flux x comme une séquence infinie de variables PPC $x \equiv \langle x_1, x_2, \ldots \rangle$. Dans ce cas, le domaine de x est $\mathcal{D}(x) = \mathcal{D}(x_1) \cdot \mathcal{D}(x_2) \cdot \ldots$, où '$\cdot$' dénote la concaténation de langages et où les domaines $\mathcal{D}(x_i)$ sont interprétés comme des alphabets. Les domaines $\mathcal{D}(x_i)$ des variables x_i sont tous supposés finis.

Cette première approche est limitante, dans la mesure où potentiellement, chaque variable x_i a un domaine distinct, et représenter une infinité de domaines est problématique. Cependant, un élément de $\mathcal{D}(x)$ est équivalent à un mot infini sur $\Sigma(x)$, en posant $\Sigma(x) = \bigcup_{k>0} \mathcal{D}(x_k)$.

Observation 2. Nous pouvons observer que, même si $\mathcal{D}(x_i)$ est fini pour tout $i > 0$, ceci n'implique pas que $\Sigma(x)$ est fini. En effet, il est possible de poser $\mathcal{D}(x_i) = [0 \ldots i]$ pour tout $i > 0$, et alors $\Sigma(x) = \bigcup_{k>0} \{1, \ldots, k\} = \mathbb{N}$ est infini alors que tous les domaines $\mathcal{D}(x_i)$ sont finis.

Comme il est plus simple de travailler avec des domaines homogènes, nous supposons que $\mathcal{D}(x_i) = \mathcal{D}(x_j)$ pour tout $i > 0$ et pour tout $j > 0$. Dans ce cas, nous pouvons poser $\Sigma(x) = \mathcal{D}(x_k)$ pour un $k \in \mathbb{N}$, et en termes de théorie des langages sur les mots infinis, $\mathcal{D}(x) = \Sigma(x)^\omega$.

Des domaines de variables flux plus structurés doivent alors être décrits par des contraintes sur les séquences, ce qui ne permet pas de retrouver l'intégralité des possibilités telles que celle évoquée dans l'observation 2, mais nous considérons ces dernières comme des cas limites qui doivent être étudiés séparément.

Définition 55 (Variable flux)**.** Une variable flux est une variable x de domaine $\mathcal{D}(x) = \Sigma(x)^\omega$, où $\Sigma(x)$ est un ensemble fini appelé **base** de x.

Étant donné un tuple $\langle x_1, \ldots, x_k \rangle$ de k variables flux et un tuple de k valeurs flux $\langle v_1, \ldots, v_k \rangle$ avec $v_i \in \mathcal{D}(x_i)$ pour $1 \leqslant i \leqslant k$, une représentation alternative consiste à considérer le tuple de variables comme une séquence unique σ d'alphabet $\Sigma(x_1) \times \cdots \times \Sigma(x_k)$, et telle que $\sigma[i] = \langle v_1[i], \ldots, v_k[i] \rangle$. Ceci montre qu'un StCSP sur k variables correspond à un StCSP sur une unique variable, ce qui permettra d'établir une correspondance utile pour effectuer le lien entre StCSPs et théorie des langages.

StCSPs Nous définissons un StCSP comme l'extension directe des CSPs aux domaines des variables flux.

Définition 56 (StCSP)**.** Un StCSP est un tuple $\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ dans lequel :

— $\mathcal{X}$ est un ensemble de variables flux

— $\mathcal{D}$ est une fonction qui associe à tout $x \in \mathcal{X}$ sont domaine $\mathcal{D}(x)$

— $\mathcal{C}$ est un ensemble de contraintes sur les variables flux

En PPC, une contrainte $c \in \mathcal{C}$ sur un tuple de k variables $\langle x_1, \ldots, x_k \rangle$ distinctes représente un sous-ensemble du produit cartésien des domaines $c \subseteq \mathcal{D}(x_1) \times \ldots \mathcal{D}(x_k)$.

De façon similaire, une contrainte sur un tuple de k variables flux $\langle x_1, \ldots, x_k \rangle$ distinctes représente un sous-ensemble du produit cartésien des domaines $c \subseteq \mathcal{D}(x_1) \times \ldots \mathcal{D}(x_k)$. Malgré cette similarité syntaxique, une contrainte PPC représente toujours un sous-ensemble fini de combinaisons de valeurs, car les domaines des variables sont toujours supposés finis. Dans le cas des variables flux, les sous-ensembles décrits par les contraintes sont en général infinis, et il devient possible de définir des contraintes indécidables, ou pour lesquelles le problème de satisfaction nécessite des algorithmes particuliers de complexités variées.

Exemple 10 (Contrainte indécidable)**.** Il est très facile de décrire une contrainte indécidable. Par exemple, en considérant une unique variable flux x de domaine $\mathcal{D}(x) = \{0, 1\}^\omega$, et en interprétant toute séquence infinie $\xi \in \mathcal{D}(x)$ comme un encodage d'un programme, la contrainte peut imposer que la séquence infinie doit correspondre à l'encodage d'un programme qui s'arrête, ce qui est impossible à mettre en œuvre.

Il est donc nécessaire de définir des classes de contraintes qui se prêtent à une résolution algorithmique. Nous définissons une classe de contraintes simples et adaptées à la programmation par contraintes dans la section suivante. Les algorithmes que nous présentons dans le chapitre suivant sont spécialement conçus pour ce type de contraintes. Bien que nous pensions qu'il existe sans doute une infinité d'autres classes de contraintes sur des variables flux qui se prêtent à une résolution algorithmique, nous n'avons pas trouvé de classe qui soient aussi naturelle que celle-ci.

6.3 Représentation de solutions

En PPC, une solution est une instanciation qui affecte à chaque variable une valeur. Ce type de solution ne pose pas de problèmes de représentation, car il suffit d'utiliser une liste finie de paires qui associe à chaque variable la valeur correspondante. Il en va de même pour la représentation d'un ensemble de solutions, qui est nécessairement fini, mais pour ces derniers, des systèmes de représentations plus compacts, tels que les MDDs [69], semblent plus attrayants.

En ce qui concerne les StCSPs, les objets manipulés sont de tailles infinies, ce qui fait que la représentation naïve, qui consiste à spécifier les valeurs des variables en extension, n'est pas réalisable. Il est toutefois possible de représenter certains objets infinis par des représentations finies.

Soient $\mathcal{R}$ un système de représentation et $\mathcal{O}$ un ensemble d'objets. Un tel système comporte un ensemble fini $\Sigma(\mathcal{R})$ de symboles, généralement soumis à certaines règles syntaxiques, et une fonction ρ qui associe à chaque représentation valide $r \in \mathcal{R}$ l'objet $\rho(r) \in \mathcal{O}$ correspondant.

Une représentation finie d'un objet $O \in \mathcal{O}$ dans $\mathcal{R}$ correspond à un mot fini $r \in \mathcal{R}^*$, et la fonction ρ a donc pour domaine $\Sigma(\mathcal{R})^*$ et pour image $\mathcal{O}$.

Une représentation de taille k est un mot de longueur k. Une borne supérieure sur le nombre N_k d'objets représentables dans $\mathcal{R}$ par des représentations de longueurs k est donnée par :

$$N_k \leqslant |\Sigma(\mathcal{R})|^k$$

et la borne supérieure sur le nombre N_* d'objets représentables par des représentations finies est donné par :

$$N_* \leqslant \sum_{k>0} N_k = \aleph_0$$

Cependant, le nombre de séquences infinie sur un alphabet fini Σ de taille supérieure à deux est donné par :

$$|\Sigma^\omega| = \aleph_1$$

Une preuve exacte de ceci utilise généralement la méthode de diagonalisation de Cantor et figure dans de nombreux ouvrages, tels que [64]. Ceci montre que le nombre de séquences infinies sur un alphabet non trivial est indénombrable, alors que le nombre de représentations finies dans un système de représentation arbitraire est infini dénombrable. Par conséquent, pour un système de représentation donné, une infinité de séquences infinies n'admettent aucune représentation finie dans ce système.

Ce résultat est à la base de la théorie de l'analyse computationelle (*computable analysis*) [149], qui étudie l'analyse réelle restreintes aux nombres représentables par des formalismes Turing-complet. Parmi les résultats, les nombres calculables forment un corps réel fermé dans lequel la relation d'égalité n'est pas décidable, mais pour lequel la relation d'ordre est calculable. Mis dans ce contexte, notre objectif devient équivalent à permettre la génération de nombres incalculables qui satisfont des contraintes spécifiées. Comme un tel nombre ne peut pas être calculé par définition, nous approximerons le problème en permettant la génération efficace d'un préfixe de longueur arbitraire d'un tel nombre, d'une manière qui permette de poursuivre efficacement la génération d'une solution aussi longtemps que souhaité.

Exemple 11 (Mots ultimements périodiques)**.** Les mots ultimement périodiques sont un système de représentation fréquemment employé pour représenter des séquences infinies. Un mot ultimement périodique est un mot de la forme uv^ω sur un alphabet Σ, avec $u \in \Sigma^*$ un mot fini et $v \in \Sigma^* \backslash \{\varepsilon\}$ un mot fini de longueur non-nulle. Pour définir formellement un système de représentation, il suffit d'augmenter l'alphabet Σ du symbole $\bullet$ supposé absent de Σ, et de représenter un mot ultimement périodique $w = uv^\omega$ par le mot fini $\rho(w) = u \cdot \bullet \cdot v$. Une propriété de ce formalisme est que tout mot ultimement périodique admet une représentation canonique de taille minimale.

Une représentation $w = uv^\omega = \rho(u \bullet v)$ n'est pas minimale si une des conditions suivantes sont satisfaites :

— Si $u = u's$ et $v = v's$ ont le même suffixe s de longueur non nulle, alors $w = u'(sv')^\omega$, $|u'| + |sv'| < |u| + |v|$ et le mot $u' \bullet sv'$ constitue une représentation de w de longueur strictement plus petite ;

— S'il existe un mot ν et un entier $k > 0$ tel que $v = \nu^k$, alors $w = \nu^\omega$, et le mot $u \bullet \nu$ constitue une représentation de w de longueur strictement plus petite

Une représentation est minimale si aucune de ces simplifications ne peut être effectuée. Tout mot ultimement périodique admet une unique représentation minimale.

Comme cela a été expliqué dans le chapitre 5, ce système de représentation est très répandu car il permet d'associer très simplement des représentation finies à des séquences infinies. Son inconvénient principal est qu'il ne permet pas de représenter des séquences infinies plus structurées, et que les séquences infinies ultimement périodiques ne sont pas toujours les plus intéressantes pour certaines applications.

Exemple 12 (Séquences infinies non périodiques)**.** Une séquence infinie particulièrement connue est la séquence σ_{TM} de Thue-Morse [105]. La définition directe est la suivante :

$$\sigma_{\mathrm{TM}}[k] = \mathrm{b}(k) \bmod 2$$

ou $b(k)$ représente le nombre de bits à 1 dans l'expansion binaire de k. La séquence de Thue-Morse est une séquence fractale, car il suffit de supprimer un élément sur deux pour obtenir à nouveau la séquence de Thue-Morse.

Le système de représentation le plus général pour les séquences infinies est naturellement celui des machines de Turing, ou de tout système Turing-équivalent, en supposant la thèse de Church-Turing vraie. Nous ne traiterons pas le problème du calcul de la machine de Turing correspondante à une séquence infinie satisfaisant des contraintes données, car nous souhaitons obtenir un système qui permet de générer n'importe quelle séquence satisfaisant des contraintes, y compris des séquences qui ne sont pas représentables, et une telle machine de Turing serait nécessairement une représentation fini d'une séquence infinie. Pour permettre la génération de séquence non représentable, une alternative consiste à manipuler des ensembles de séquences.

Représentation d'ensembles de séquences infinies Les systèmes de représentation d'ensembles de séquences infinies sont soumis aux mêmes limitations en terme de représentabilité, mais offrent l'avantage de permettre d'extraire des séquences infinies qui n'admettent pas de représentation finie.

L'idée principale consiste à exploiter le non-déterminisme pour conduire un programme à effectuer une infinité de choix arbitraires. Le nombre des combinaisons de choix possibles effectués au cours d'une génération devient alors indénombrable, ce qui permet la génération de séquences non-représentables.

N'importe quel système computationnel peut être utilisé à ces fins. Nous distinguons deux types de systèmes, en fonction de la quantité de mémoire utilisée par ceux-ci. Une machine de Turing, ou un système équivalent, utilise une mémoire non-bornée, ce qui a pour effet que la quantité de mémoire utilisée peut croître indéfiniment. Dans un contexte applicatif, la mémoire des systèmes utilisés étant toujours finie, ceux-ci imposent artificiellement une borne supérieure sur la longueur des séquences générées, car le programme finira nécessairement par utiliser plus de mémoire qu'il n'est disponible, où alors la séquence particulière de choix qui aura été faite aurait tout aussi bien pu être produite par un système à mémoire finie.

Les systèmes à mémoire finie sont équivalent à des automates finis. Les automates de Büchi, introduits dans le chapitre 2 et discutés chapitre 4, sont des exemples de tels automates, qui permettent de construire des représentations finies d'ensembles de séquences infinies. Comme tout automate de Büchi dont le langage est non vide admet au moins un mot ultimement périodique [133], un automate de Büchi contient nécessairement des mots représentables, et il en va de même pour tout système à mémoire finie. Mais ces automates permettent également de générer des mots apériodiques dénués de toute structure globale, mais satisfaisant des contraintes locales établies par la structure de la fonction de transition de l'automate.

Nous aurons donc comme objectif de rechercher des représentations finies d'ensembles de solutions pour les StCSPs, puis de nous servir de ces représentations pour générer des solution particulières à la demande. Cette approche se distingue des approches à base de contraintes existantes, telles que celles décrites dans le chapitre 5, car celles-ci ont pour objectif la recherche de représentation finies de solutions particulières, généralement basées sur les mots ultimement périodiques.

Optimisation. L'optimisation est un thème habituel en PPC, car comme les instances admettent en général plusieurs solutions, il est parfois naturel d'en considérer certaines comme meilleures que d'autres. Comme l'espace de recherche est nécessairement fini, il est toujours possible de trouver la solution optimale en calculant toutes les solutions et en gardant celle qui minimise ou maximise la fonction objectif.

Dans le cas des variables flux, la situation est différente, car l'ensemble des solutions d'un problème peut être infini, et il devient alors possible de définir des problèmes d'optimisations pour lesquels l'optimum n'existe pas.

Exemple 13. Soit x une variable flux de base $\Sigma(x) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, et f la fonction d'optimisation suivante :

$$f(\xi) = \sum_{n>0} \xi[n] \times 10^{-n}$$

L'objectif est de minimiser la valeur de f avec la contrainte que $f(\xi) \neq 0$. Cette fonction assigne à chaque flux $\langle \xi[1], \xi[2], \ldots \rangle$ le nombre réel $0, \xi[1]\xi[2]\ldots$. La contrainte sur f peut s'exprimer par l'obligation qu'un chiffre différent de 0 apparaisse au moins une fois.

Comme pour tout nombre réel $r > 0$, il existe un nombre réel $r' > 0$ tel que $r > r'$, pour tout flux ξ, il est possible de trouver un flux ξ' tel que $f(\xi') < f(\xi)$. Donc pour toute solution, il

existe une solution qui est meilleure du point de vue du critère d'optimisation, ce qui implique que l'optimum de cette fonction n'existe pas.

Certains critères d'optimisation qui paraissent intuitifs nécessitent d'être examinés plus en détail. Par exemple, soit la famille de mots infinis $\langle w_k \rangle_{k \in \mathbb{N}}$ telle que $w_k = a^k b$, et $w = w_1 \cdot w_2 \cdot \ldots \cdot w_k \cdot \ldots$ la concaténation de tous les mots de $\langle w_k \rangle_{k \in \mathbb{N}}$. Un préfixe de ce mot est le mot $u = babaabaaabaaaabaaaaab\ldots$, et il serait naturel d'exprimer le fait que les 'a' sont en plus grande proportion que les 'b' dans w. Cependant, les positions de w auxquelles correspondent des 'b' sont en bijection avec les entiers naturels, et il en va de même pour les 'a', suite à quoi il est nécessaire de déduire qu'il y a autant de 'a' que de 'b' dans w. Sur les mots infinis, tous les symboles qui apparaissent une infinité de fois sont en proportion identiques, et dans un contexte général, il y a donc peut de sens à "minimiser" les nombres d'occurences de certains par rapport à d'autres. Seuls les symboles qui apparaissent un nombre fini de fois peuvent être sujets à des critères quantifatifs.

Malgré cela, si les considérations ne portent que sur des mots ultimement périodiques, alors tout mot w admet une représentation minimale $w = uv^\omega$, et il devient possible de s'intéresser aux proportions des symboles de v. Par exemple, si $w = (abbc)^\omega = v$, alors il est naturel de considérer que w est constitué d'une moitié de 'b', d'un quart de 'a' et d'un quart de 'c', alors que du point de vue précédent, tous les symboles sont présents en quantités identiques. Il est toutefois possible de définir dans ce contexte des critères d'optimisation qui ne permettent pas de calculer un optimum. Par exemple, soit le langage ω-régulier $\mathcal{L} = (ab^*c)^\omega$. Alors si l'objectif est de trouver un mot ultimement périodique de partie périodique $v = u(ab^kc)^\omega$ de $\mathcal{L}$ qui maximise la proportion de 'b', pour tout $k \in \mathbb{N}$, le mot $(ab^{k+1}c)^\omega$ est meilleur que le mot $(ab^kc)^\omega$ par rapport à ce critère, et l'optimum n'existe pas.

Ce type de situation n'existe pas en PPC. Même dans le cas de la programmation linéaire sur les nombres réels, comme les objets manipulés sont des nombres flottants, c'est-à-dire des approximations de tailles finies des nombres réels, le problème est virtuellement inexistant. Les questions d'optimisation sont donc plus délicates en programmation par contraintes sur les flux.

6.4 Contraintes de voisinages

Nous proposons les contraintes de voisinages afin d'exploiter les contraintes PPC ainsi que les algorithmes de propagation existants dans le contexte des flux. L'objectif de notre formalisme consiste à pouvoir structurer des séquences infinies avec des contraintes de telle sorte que celles-ci portent sur l'intégralité des séquences. Le principe que nous utilisons consiste à appliquer une contrainte sur un petit voisinage, puis à répliquer les contraintes indéfiniment le long de la séquence.

Afin d'étendre la portée des contraintes PPC sur des variables à des instants différents, nous utilisons la notion de terme introduite dans le chapitre 5, car notre formalisme correspond à une logique temporelle sur des domaines finis.

Définition 57 (Terme)**.** Soit x une variable flux. Un **terme** est une variable flux, ou l'application de l'opérateur '**X**' à un terme.

Les contraintes de voisinage sont des contraintes PPC qui portent sur des termes, et qui sont interprétées selon une sémantique qui correspond à celle des interprétations temporelles du chapitre 5.

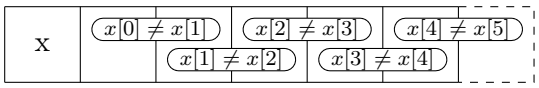

FIGURE 6.1 – Diagramme correspondant à la contrainte $x \neq \mathbf{X}x$.

Définition 58 (Contrainte de voisinage)**.** Une **contrainte de voisinage** d'un StCSP $\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ est un tuple $C = \langle \mathfrak{c}, T \rangle$ dans lequel $\mathfrak{c}$ est une contrainte PPC d'arité k, et $T = \langle T_1, \ldots, T_k \rangle$ est un tuple de k termes, où pour tout $1 \leqslant i \leqslant k$, $T_i = \mathbf{X}^{t_i} x_i$ avec $t_i \geqslant 0$ et $x_i \in \mathcal{X}$. La portée d'une contrainte de voisinage est l'ensemble des variables flux sur lesquelles portent les termes sur lesquels la contrainte porte : $\mathrm{var}(C) = \{x_1, \ldots, x_k\}$.

Une contrainte de voisinage s'applique sur l'intégralité des séquences.

Définition 59 (Satisfaction d'une contrainte de voisinage)**.** Soient $\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un StCSP, $c = \langle \mathfrak{c}, T \rangle$ une contrainte de voisinage d'arité k de $\mathcal{C}$, avec $T = \langle \mathbf{X}^{t_1} x_1, \ldots, \mathbf{X}^{t_k} x_k \rangle$, et pour tout $1 \leqslant i \leqslant k$, $t_i \geqslant 0$ et $x_i \in \mathcal{X}$. Soient $I : \mathrm{var}(C) \to \mathbb{Z}^\omega$ une instanciation des variables qui associe à tout $x \in \mathrm{var}(c)$ une séquence de $\mathcal{D}(x)$. Alors I satisfait c si, pour tout $n > 0$:

$$\langle I(x_1)[n + t_1], \ldots, I(x_k)[n + t_k] \rangle \vDash \mathfrak{c}$$

Exemple 14 (Exemple de contrainte de voisinage)**.** La figure 6.1 illustre un exemple de contrainte de voisinage sur une variable flux x de domaine arbitraire. La contrainte $x \neq \mathbf{X}x$ impose que la valeur de x change à chaque instant. Si le domaine de base de x est un singleton, alors la contrainte n'a pas de solution. Si $|\Sigma(x)| = 2$, alors la contrainte admet uniquement deux solutions. Si $|\Sigma(x)| > 2$, alors la contrainte admet une infinité indénombrable de solutions. Par exemple, si $\Sigma(x) = \{1, 2\}$, alors les deux solutions sont $x = (12)^\omega$ et $x = (21)^\omega$.

Temporalité. Un paramètre important des StCSP est leur temporalité, c'est à dire le décalage maximal effectué sur les variables.

Définition 60 (Temporalité)**.** La **temporalité** d'un terme $\mathbf{X}^t x$ est t. La temporalité d'une contrainte de voisinage est la temporalité maximale des termes sur lesquels elle porte. La temporalité d'un StCSP est la temporalité maximale de ses contraintes. Enfin la temporalité d'une variable d'un StCSP est la temporalité maximale de tous les termes qui portent sur cette variable dans toutes les contraintes.

Par exemple, la temporalité du StCSP de la figure 14 est 1. Nous développons dans le chapitre suivant des algorithmes pour la résolution de StCSP. Pour ces algorithmes, les StCSPs de temporalités 1 jouent un rôle prépondérant.

Chapitre 7

Résolution

Sommaire

Dans ce chapitre, nous présentons des algorithmes de résolution des StCSPs. Nous présentons d'abord une première méthode qui permet de calculer l'intégralité du système de transitions via la résolution d'un unique CSP. Nous montrons ensuite que cette méthode peut-être modifiée pour obtenir une résolution des StCPS par parcours en profondeur du système de transitions via la résolution d'une multitude de CSPs. Enfin, nous comparons ces deux méthodes, qui seront illustrées sur des exemples dans le chapitre 8.

7.1 Première méthode

La première méthode permet de résoudre les StCSPs de temporalité 1 en respectant les hypothèses du chapitre précédent, c'est-à-dire en particulier en résolvant un unique CSP. Nous montrons qu'à tout StCSP de temporalité 1 correspond un système de transitions, et que les transitions de ce systèmes correspondent aux solutions d'un CSP.

En ce qui concerne les StCSPs de temporalités supérieures, nous montrons que ceux-ci sont équivalent à des StCSPs de temporalité 1, qui sont obtenus par une procédure dite de "réduction de temporalité".

Le principe de la méthode consiste donc d'abord à réduire la temporalité d'un StCSP, puis à calculer le système de transitions associé par résolution d'un CSP, et enfin à parcourir ce système pour générer une solution. Cette approche implique le calcul de l'intégralité du système de transitions, qui peut être très grand, mais permet de générer efficacement des solutions une fois ce pré-calcul effectué.

7.1.1 Réduction de temporalité

La méthode de réduction de temporalité consiste à ajouter des variables auxiliaires qui représentent les valeurs de certaines variables à des des instants futurs, et à les relier aux variables qu'elles représentent par l'intermédiaire de contraintes d'égalité de temporalité unité.

Soit $\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un StCSP de temporalité k. Alors il existe dans $\mathcal{S}$ au moins une contrainte $c \in \mathcal{C}$ de temporalité k, qui porte sur un terme de $\mathbf{X}^k x$ avec $x \in \mathcal{X}$.

Soit x' une nouvelle variable flux telle que $x' \notin \mathcal{X}$. La contrainte d'égalité $x' = \mathbf{X}x$ permet d'exprimer que x' est identique à la variable x aux instants ultérieures. Par exemple, si la valeur de x est $\langle 1, 2, 3, 2, 1, 2, 3, 2, 1, \ldots \rangle$, alors la valeur de x' est nécessairement $\langle 2, 3, 2, 1, 2, 3, 2, 1, \ldots \rangle$.

Par ailleurs, du fait de la contrainte d'égalité, $\mathbf{X}^k x$ est équivalent à $\mathbf{X}^{k-1} x'$. En substituant tous les termes de la forme $\mathbf{X}^k x$ par $\mathbf{X}^{k-1} x'$ dans toutes les contraintes, la temporalité de la variable x est réduite de 1. Soit $\mathcal{S}' = \langle \mathcal{X}', \mathcal{D}', \mathcal{C}' \rangle$ le StCSP résultant de cette substitution, dans lequel $\mathcal{X}' = \mathcal{X} \cup \{x'\}$, $\mathcal{D}(x') = \mathcal{D}(x)$, et C' contient la contrainte $x' = \mathbf{X}x$, ainsi que les contraintes dans lesquelles les termes de temporalité k portant sur x ont été remplacés par des termes de temporalité $k-1$ portant sur x'. Si x est la seule variable de $\mathcal{S}$ de temporalité k, alors la temporalité de $\mathcal{S}'$ est nécessairement de temporalité $k' < k$, et la temporalité de $\mathcal{S}$ est donc réduite.

$\mathcal{S}$ et $\mathcal{S}'$ sont équivalents dans le sens où toute solution de $\mathcal{S}'$ projetée sur les variables de $\mathcal{S}$ est une solution de $\mathcal{S}$, car du fait de la contrainte d'égalité, les termes $\mathbf{X}^k x$ et $\mathbf{X}^{k-1} x'$ sont indistingables.

Comme le nombre de variables flux d'un StCSP est fini, le nombre de variables flux de temporalité maximale est nécessairement fini. Si $\mathcal{S}$ contient m variables de temporalité k, alors m applications de la procédure précédente produiront un StCSP équivalent de temporalité $k-1$ contenant m variables auxiliaires supplémentaires et m contraintes d'égalité supplémentaires.

Par un argument similaire, comme la temporalité d'un StCSP est nécessairement finie, un nombre fini d'applications de la procédure de substitution est nécessaire pour obtenir une StCSP de temporalité 1.

Exemple 15. Nous illustrons la procédure de réduction de temporalité sur un exemple. Soit $\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, dans lequel :
- $\mathcal{X} = \{x, y\}$;
- $\mathcal{D}(x) = \mathcal{D}(y) = \mathbb{B}$;
- $\mathcal{C} = \{x = \mathbf{X}y \vee \mathbf{X}^2 x, y = \mathbf{X}^3 x\}$.

Cet exemple utilise des contraintes fonctionnelles avec références aux valeurs futures. De nombreux formalismes temporels n'introduisent que la notion de valeurs futures fonction de valeurs passées [121]. Notre formalisme permet de généraliser cette notion en introduisant des variables dont les valeurs sont fonctions de variables à des instants futurs, qui n'ont conceptuellement pas encore été calculées. Une telle fonction est un cas particulier de la notion de contrainte.

La procédure de réduction de temporalité substitue le terme $\mathbf{X}x'$ à $\mathbf{X}^2 x$, en introduisant la variable x' de domaine $\mathcal{D}(x') = \mathcal{D}(x)$ et la contrainte d'égalité $x' = \mathbf{X}x$. Le terme $\mathbf{X}^3$ nécessite l'introduction d'une variable x'' soumise à la contrainte $x'' = \mathbf{X}x'$, de domaine $\mathcal{D}(x'') = \mathcal{D}(x')$. Le StCSP de temporalité 1 équivalent $\mathcal{S}' = \langle \mathcal{X}', \mathcal{D}', \mathcal{C}' \rangle$ est le suivant :
- $\mathcal{X}' = \{x, y, x', x''\}$;
- $\forall x \in \mathcal{X}', \mathcal{D}(x) = \mathbb{B}$;
- $C = \{x = \mathbf{X}y \vee \mathbf{X}x', y = \mathbf{X}x'', x' = \mathbf{X}x, x'' = \mathbf{X}x'\}$.

7.1.2 Système de transitions associé à un StCSP de temporalité 1

Dans cette section, nous montrons qu'à tout StCSP $\mathcal{S}$ de temporalité 1 correspond un système de transitions $\mathcal{T}(\mathcal{S})$ pour lequel tout chemin de $\mathcal{T}(\mathcal{S})$ est une solution de $\mathcal{S}$ et toute solution de $\mathcal{S}$ est un chemin dans $\mathcal{T}(\mathcal{S})$.

Définition 61 (Système de transitions)**.** Un **système de transitions** est un tuple $\mathcal{T} = \langle \mathcal{Q}, \Sigma, \delta \rangle$ dans lequel :

— $\mathcal{Q}$ est un ensemble fini d'états ;

— $\delta \in \mathcal{Q} \times \mathcal{Q}$ est une relation de transitions

Un séquence infinie d'état $\xi \in \mathcal{Q}^\omega$ est **acceptée** par $\mathcal{T}$ si pour tout $k > 0$, $\langle \xi[k], \xi[k+1] \rangle \in \delta$.

Soit $\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un StCSP de temporalité 1, et soit $\mathcal{T}(\mathcal{S}) = \langle \mathcal{Q}, \delta \rangle$ le système de transitions associé à $\mathcal{S}$.

Nous définissons $\mathcal{T}(\mathcal{S})$ de façon déclarative en utilisant les contraintes de $\mathcal{S}$. Comme $\mathcal{S}$ est de temporalité 1, les contraintes de $\mathcal{S}$ sont soit de temporalité 0, soit de temporalité 1. Nous distinguons trois types de contraintes :

Contraintes instantanées. Dans une contrainte instantanée, tous les termes sur lesquels elle porte sont de temporalité 0 ;

Contraintes mixtes. Dans une contrainte mixte, certains termes sont de temporalité 0 et d'autres termes sont de temporalité 1 ;

Contraintes singulières. Dans une contrainte singulière, tous les termes sont de temporalité 1.

Les contraintes instantanées de $\mathcal{S}$ définissent les propriétés qui doivent être vraies à chaque instant. Ces contraintes permettent de définir l'ensemble des états de $\mathcal{T}(S)$. En notant $\mathcal{C}^0$ les contraintes instantanées de $\mathcal{S}$, l'ensemble des états $\mathcal{Q}$ de $\mathcal{T}(S)$ est défini par :

$$\mathcal{Q} = \{I \in \mathcal{I}_\Sigma(\mathcal{X}) \mid I \vDash \mathcal{C}^0\}$$

Commes les contraintes mixtes contiennent des termes de temporalité 0 et des termes de temporalité 1, elles modélisent naturellement les transitions du système. En ce qui concerne les contraintes singulières, elles représentent en réalité des contraintes instantanées, mais qui ne s'appliquent pas à l'instant initial. Il est donc impossible de les utiliser pour définir l'ensemble des états, et elles doivent être prises en compte dans la définition du système de transitions.

Pour discuter des transitions, nous observons qu'une transition de δ est isomorphique à une instanciation de termes. Soit $\mathcal{X}_\mathbf{X}$ l'ensemble de termes suivants :

$$\mathcal{X}_\mathbf{X} = \bigcup_{x \in \mathcal{X}} \{x, \mathbf{X}x\}$$

et soit $\Sigma_\mathbf{X} : \mathcal{X}_\mathbf{X} \to \mathrm{Img}(\Sigma)$ la fonction qui associe à un terme le domaine de la variable sous-jacente, où 'Img' est la fonction qui associe à une fonction son image, et Σ est la fonction qui associe à une variable $x \in \mathcal{X}$ sa base :

$$\Sigma_\mathbf{X}(t) = \begin{cases} \Sigma(x) & \text{si } t = x \\ \Sigma(x) & \text{si } t = \mathbf{X}x \end{cases}$$

Soit $\iota : \mathcal{I}_\Sigma(\mathcal{X}) \times \mathcal{I}_\Sigma(\mathcal{X}) \to \mathcal{I}_{\Sigma_\mathbf{X}}(\mathcal{X}_\mathbf{X})$ la fonction qui associe à une paires d'instanciations $\langle I, I' \rangle$ l'instanciation $I_\mathbf{X} \in \mathcal{I}_{\Sigma_\mathbf{X}}(\mathcal{X}_\mathbf{X})$ suivante :

$$I_\mathbf{X}(t) = \begin{cases} I(x) & \text{si } t = x \\ I'(x) & \text{si } t = \mathbf{X}x \end{cases}$$

En notant $\mathcal{C}^1 = \mathcal{C} \backslash \mathcal{C}^0$, la bijection ι permet d'utiliser directement les contraintes pour définir la fonction de transition :

$$\delta = \{\iota^{-1}(I_\mathbf{X}) \mid \mathcal{I}_\mathbf{X} \in \mathcal{I}_{\Sigma_\mathbf{X}}(\mathcal{X}_\mathbf{X}) \wedge I_\mathbf{X} \vDash \mathcal{C}^1\}$$

7.1.3 Algorithme

Nous présentons dans cette section un algorithme pour la résolution de StCSPs. La fonction polymorphique T associe à un StCSP, une contrainte ou une variable sa temporalité. La notation $c[i]$ dénote le i-ème terme d'une contrainte c, et la notation $|c|$ dénote le nombre de termes de c. La fonction `Reduire` de l'algorithme 7.1.3 transforme un StCSP de temporalité k en un StCSP équivalent de temporalité 1. Il effectue simplement les substitutions des termes de temporalité maximales telles que décrites dans la sous-section 7.1.1 tant que la temporalité du StCSP est supérieure à 1. Ceci consiste à chaque itération à introduire une variables auxiliaire et une contrainte d'égalité, et à effectuer les substitutions correspondantes.

Algorithme 7.1 Réduction de temporalité

```
1: Fonction Reduire(!S = ⟨X, D, C⟩ : StCSP)
2:   tant que T(S) > 1 faire
3:     k = T(S)
4:     pour chaque x ∈ {x ∈ X | T(x) = T(S)} faire
5:       Ajouter(X, x')
6:       Ajouter(C, x' = Xx)
7:       pour chaque c ∈ C faire
8:         pour i ∈ [1 ... |c|] faire
9:           si c[i] = X^k x alors
10:            c[i] = X^{k-1} x'
11:          fin si
12:        fin pour
13:      fin pour
14:    fin pour
15:  fin tant que
16: Fin fonction
```

La résolution d'un StCSP $\mathcal{S}$ consiste à calculer le système de transitions $\mathcal{T}(\mathcal{S})$ associé à $\mathcal{S}$ en résolvant un problème de satisfaction de contraintes $\mathcal{P}(\mathcal{S})$. Les principes de la résolution des CSPs ont été décrits dans le chapitre 3. Nous présentons la procédure de génération de $\mathcal{P}(\mathcal{S})$ dans l'algorithme 7.2. Nous notons c^{+1} une contrainte de temporalité 0 appliquée aux termes correspondant de temporalité 1, c'est-à-dire que si c s'applique sur les termes $\langle t_1, \ldots, t_k \rangle$, alors c^{+1} est la même contrainte c appliquée aux termes $\langle \mathbf{X}t_1, \ldots, \mathbf{X}t_k \rangle$.

La fonction 7.2 prend en entrée un StCSP $\mathcal{S}$, et réduit sa temporalité à la ligne 3 par un appel à la fonction `Reduire` de l'algorithme . À la ligne 4, la variable $\mathcal{X}_\mathcal{P}$ représente l'ensemble des termes de temporalités 0 et 1, qui sont les variables de $\mathcal{P}(\mathcal{S})$. Leurs domaines sont initialisées dans les lignes 5 à 8 à partir des bases des variables de $\mathcal{S}$. Dans les lignes 9 à 16, les contraintes de $\mathcal{P}$ sont imposées à partir de celles de $\mathcal{S}$. Les contraintes instantanées sont appliquées de part et d'autres des transitions, et les autres contraintes (mixtes et singulières) sont appliquées sans modification, puisque les variables de $\mathcal{P}$ sont directement des termes.

Algorithme 7.2 Calcul du CSP associé à un StCSP

1: **Fonction** `CalculCSP`($!\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: `StCSP`)
2: $\mathcal{P} = \langle \mathcal{X}_\mathcal{P}, \mathcal{D}_\mathcal{P}, \mathcal{C}_\mathcal{P} \rangle$: `CSP`
3: `Reduire`($\mathcal{S}$)
4: $\mathcal{X}_\mathcal{P} = \bigcup_{x \in \mathcal{X}} \{x, \mathbf{X}x\}$
5: **pour** $x \in \mathcal{X}$ **faire**
6: $\mathcal{D}_\mathcal{P}(x) = \Sigma(x)$
7: $\mathcal{D}_\mathcal{P}(\mathbf{X}x) = \Sigma(x)$
8: **fin pour**
9: **pour chaque** $c \in \mathcal{C}$ **faire**
10: **si** $T(c) = 0$ **alors**
11: `Ajouter`($\mathcal{C}_\mathcal{P}, c$)
12: `Ajouter`($\mathcal{C}_\mathcal{P}, c^{+1}$)
13: **sinon**
14: `Ajouter`($\mathcal{C}_\mathcal{P}, c$)
15: **fin si**
16: **fin pour**
17: **renvoyer** $\mathcal{P}$
18: **Fin fonction**

Le CSP $\mathcal{P}(\mathcal{S})$ peut être utilisé de deux manières pour générer des solutions. Nous présentons un algorithme de génération du système de transitions $\mathcal{T}(\mathcal{S})$ dans l'algorithme 7.3. Le paramètre d'entrée est le CSP $\mathcal{P}(\mathcal{S})$ calculé dans 7.2.

Le systèmes de transitions est introduit à la ligne 2, puis dans les lignes 3 à 12, chaque solution de $\mathcal{P}$ est utilisée pour générer une transition. Les paires d'instanciations sont générées dans les lignes 5 à 8, et l'ensemble d'états est construit aux lignes 9 et 10. En effet, le CSP $\mathcal{P}(\mathcal{S})$ ne donne pas directement l'ensemble des états, est celui-ci doit être déduit de l'ensemble des transitions. Il serait possible de calculer l'ensemble d'états directement à partir des contraintes instantanées, mais ceci nécessiterait la résolution redondante d'un deuxième CSP.

L'algorithme 7.3 n'est pas directement utilisable pour générer des solutions d'une façon efficace, car il est possible que celui-ci contienne des impasses, qui obligerait un parcours aléatoire à revenir en arrière. La suppression des impasses peut être réalisée efficacement par un parcours en profondeur. En *model checking*, le problème est résolu en ajoutant des cycles à toutes les impasses pour obtenir des structures de Kripke. Cette opération engendrerait ici un système de transitions qui ne satisferait pas les contraintes, car si un état I de $\mathcal{T}(S)$ est une impasse, c'est parce que la transition $I \to I$ ne satisfait pas les contraintes. Un tel état ne permet que de générer des solutions finies, donc le supprimer ne change pas l'ensemble des solutions produites par le système de transitions.

Algorithme 7.3 Calcul du système de transitions associé à un StCSP

```
1: Fonction CalculTS(S = ⟨X, D, C⟩ : StCSP, P = ⟨X_P, D_P, C_P⟩ : CSP)
2:   T = ⟨Q, δ⟩ : TS
3:   pour chaque I_X ∈ Sol(P) faire
4:     I, I' : I(X)
5:     pour chaque x ∈ X faire
6:       I(x) = I_X(x)
7:       I'(x) = I_X(Xx)
8:     fin pour
9:     Ajouter(Q, I)
10:    Ajouter(Q, I')
11:    Ajouter(δ, ⟨I, I'⟩)
12:  fin pour
13:  renvoyer T
14: Fin fonction
```

Nous donnons dans l'algorithme une description déclarative de cette procédure. La boucle 2–9 est exécutée tant qu'il existe un état qui n'est la source d'aucune transition. Dans ce cas, il faut supprimer cet état (ligne 3), mais aussi supprimer toutes les transitions qui mènent vers cet état (lignes 4 à 8).

Algorithme 7.4 Suppression des impasses d'un systèmes de transitions

```
1: Fonction SupprimerImpasses(!T = ⟨Q, δ⟩ : TS)
2:   tant que ∃q ∈ Q, ∀q' ∈ Q, ¬δ(q, q') faire
3:     Supprimer(Q, q)
4:     pour chaque d ∈ δ faire
5:       si d[2] = q alors
6:         Supprimer(δ, d)
7:       fin si
8:     fin pour
9:   fin tant que
10: Fin fonction
```

7.2 Deuxième méthode

L'approche utilisée dans la première méthode nécessite de générer l'intégralité du système de transitions avant de pouvoir commencer à générer des solutions. Cette approche est intéressante pour de petits problèmes, car elle permet d'exploiter au maximum les algorithmes de propagation. L'inconvénient principal est que ce procédé nécessite de calculer toutes les solutions d'un CSP, ce qui est exponentiel en espace, et donc en temps, dans le pire des cas.

Nous proposons ici une méthode qui est moins efficace en terme de propagation, mais qui permet de générer des solutions pour des problèmes de tailles plus importantes. Le principe consiste à utiliser la programmation par contraintes pour générer un parcours en profondeur du système de transitions. L'algorithme présenté ici reste générique, dans la mesure où il peut être adapté à la recherche de différents objets, telle que des solutions ultimement périodiques, ou des

composantes fortement connexes non-déterministes. Ce dernier type de résultat permet de ne calculer qu'une partie du système de transitions, mais tout de même de générer des solutions non ultimement périodiques.

La méthode est basée sur le CSP calculé par l'algorithme 7.2. Le principe consiste d'abord à chercher une solution de ce CSP, qui donne une première transition satisfaisant les contraintes. En utilisant la destination de cette transition pour fixer les valeurs des termes de temporalité 0, une résolution du même CSP permet d'obtenir une autre transition satisfaisant les contraintes de temporalité 1.

Le problème des impasses peut être géré de plusieurs façons. Une première façon consiste simplement à effectuer un retour arrière lorsqu'une transition se révèle être une impasse. Si le procédé de résolution n'est pas aléatoire, les solutions pourront toutes être examinées, et de cette façon, les états qui mènent vers des impasses pourront être identifiés. Une optimisation simple consiste à intégrer ces derniers dans une table de hachage. Comme le nombres d'états menant vers des impasses est exponentiellement grand, une structure de données plus compacte peut se révéler utile. Les MDDs [70], qui sont des généralisations des BDDs [23] à des domaines multivaluées, sont des structures de données qui semblent adaptées à cet usage.

Comme plusieurs solutions existent pour gérer ce problème, nous l'ignorons dans l'algorithme 7.5. Celui-ci enregistre les transitions trouvées au fur et à mesure dans une file F. Sans conditions supplémentaires, il est impossible de décider de générer la solution en cours de calcul, car il est impossible de déterminer avec certitude si le dernier état mène à une impasse. En utilisant une condition qui permet de générer des morceaux de cycles, on pourrait vider la file dès que le dernier état à déjà été visité et reprendre la résolution en partant de cet état, mais si le processus de résolution est déterministe et que les paramètres heuristiques sont inchangées, la partie suivante sera strictement identique à la première. Il faudrait procéder à la résolution en utilisant des paramètres différents à chaque fois pour obtenir des solutions plus intéressantes. Nous cherchons donc simple des solutions ultiment périodiques ici, en gardant en mémoire que ce sont les variations sur ce thème qui nous intéressent.

Dans l'algorithme 7.5, la fonction `ResoudreEnLigne` prend en entrée un StCSP $\mathcal{S}$ pour avoir accès à ses variables, le CSP $\mathcal{P}(\mathcal{S})$ associé à $\mathcal{S}$ (variable $\mathcal{P}_{\mathbf{X}}$) calculé par la fonction `CalculCSP` de l'algorithme 7.2, et une file F. Le résultat R est initialisé à $\perp$, qui représente un objet sans valeur. La boucle principale examine chaque transition t solution de $\mathcal{P}_{\mathbf{X}}$, et tente de prolonger en contruisant un nouveau CSP dans lequel les termes de temporalité 0 sont contraints à être égaux à ceux de temporalité 1 dans t. Les paires d'instanciations sont construites dans les lignes 5 à 8. Le cas d'une file vide signal le début de la procédure (ligne 9). Dans le cas générique (lignes 12 à 26), la présence d'une instanciation déjà produite indique qu'une solution ultiment périodique a été trouvée (lignes 12 à 14). Dans le cas contraire, un appel récurisf à `ResoudreEnLigne` est effectué. Si cet appel échoue, cela signifie que le préfixe courant décrit par F ne permet pas de produire une solution infinie, et qu'il est nécessaire de continuer l'ensemble de solutions courant. Si toutes les solutions ont été essayés, alors la boucle renvoit $\perp$ pour signifier l'absence de solution courante (ligne 28). Si le premier appel renvoit $\perp$, c'est qu'il n'existe pas de solutions, sinon, c'est que le préfixe courant ne permet pas d'obtenir une solution. La valeur de la variable R est différente de $\perp$ seulement si la ligne 14 a été executée, ce qui se propage récursivement à la ligne 20. Une solution est une paire $\langle F, k\rangle$ dans laquelle F est une séquence d'instanciations et k l'indice dans cette séquence qui détermine le cycle.

Algorithme 7.5 Génération de solutions par un parcours en profondeur

1: **Fonction** `ResoudreEnLigne`($\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: `StCSP`,$\mathcal{P}_{\mathbf{X}} = \langle \mathcal{X}_{\mathbf{X}}, \mathcal{D}_{\mathbf{X}}, \mathcal{C}_{\mathbf{X}} \rangle$: `CSP`, F : `File`)
2: $R = \bot$
3: **pour chaque** $I_{\mathbf{X}} \in \mathrm{Sol}(\mathcal{P}_{\mathbf{X}})$ **faire**
4: $I, I' : \mathcal{I}(\mathcal{X})$
5: **pour chaque** $x \in \mathcal{X}$ **faire**
6: $I(x) = I_{\mathbf{X}}(x)$
7: $I'(x) = I_{\mathbf{X}}(\mathbf{X}x)$
8: **fin pour**
9: **si** $F = \emptyset$ **alors**
10: `Empiler`(F, I)
11: **fin si**
12: **si** $I' \in F$ **alors**
13: $k =$ `IndiceDe`(F, I') — Indice de I' dans F
14: **renvoyer** $\langle F, k \rangle$
15: **sinon**
16: $\mathcal{P}'_{\mathbf{X}} = \mathcal{P}_{\mathbf{X}}$
17: **pour chaque** $x \in \mathcal{X}$ **faire**
18: $\mathcal{D}(x) = I'(x)$
19: **fin pour**
20: $R =$ `ResoudreEnLigne`$(\mathcal{S}, \mathcal{P}'_{\mathbf{X}}, F)$
21: **si** $R = \bot$ **alors**
22: `Depiler`(F)
23: **sinon**
24: **renvoyer** R
25: **fin si**
26: **fin si**
27: **fin pour**
28: **renvoyer** $\bot$
29: **Fin fonction**

7.3 Discussion

La première méthode permet d'obtenir l'intégralité du système de transitions par résolution d'un unique StCSP, mais impose de calculer entièrement ce système avant de pouvoir commencer la génération d'une solution, ce qui est réalisable seulement pour des problèmes de tailles modestes.

La deuxième méthode est plus flexible, mais nécessite la résolution d'une multitude de CSPs. Cependant, elle ne peut pas être utilisée pour obtenir des solutions de façon fluide, car la constitution et la résolution de nouveaux CSPs nécessitent beaucoup plus de ressources que la manipulation d'un unique CSP. Mais surtout, il est impossible d'empêcher le parcours en profondeur de s'engager dans des impasses, ce qui oblige l'algorithme à effectuer des retours arrières, et à garder un mémoire des parties d'une solutions jusqu'à ce qu'un cycle soit trouvé. Plutôt que de s'arrêter au premier au premier cycle, il est techniquement possible de continuer la résolution sur un autre cycle, puis d'oublier le cycle précédent une fois ce nouveau cycle trouvé, mais ceci nécessite que l'ordre dans lequel les solutions sont produites ne soit pas constant, sans quoi le prochain cycle trouvé sera identique au précédent. Il est également possible de mettre à profit le temps utilisé pour traiter la dernier partie cyclique trouvée pour calculer la partie suivante. Mais les résultats resteront très probablement peu fluide si le nombre d'impasses rencontrés est trop grand.

Nous verrons dans les exemples que les techniques de résolution des StCSPs ainsi que la présentation de solutions dépendent beaucoup du type de problème considéré. Si le nombre de composante fortement connexes est petit, alors il serait possible de laisser l'utilisateur choisir celles que celui-ci préfère, mais si le nombre de composantes fortement connexes est trop important, il devient difficile de toutes les présenter à l'utilisateur. Il est également possible que certaines composantes connexes soient considérées plus intéressantes que les autres. Par exemple, une composante fortement connexe d'une dizaine de nœuds sera probablement plus intéressante qu'une composante de deux nœuds, car elle permettra une plus grand diversité dans la forme des solutions obtenues. Cependant, une composante fortement connexe complète de plusieurs nœuds sera sûrement moins intéressante qu'une composante pas complète car elle sera équivalente à l'aléatoire.

Chapitre 8

Exemples

Sommaire

Ce chapitre présente trois exemples de StCSPs de tailles variées. Le premier exemple porte sur un problème de gestion de traffic urbain. Cet exemple extrêmement petit permet d'illustrer en détails le procédé de résolution des StCSP. Un exemple d'un intérêt plus pratique porte sur l'ordonnancement d'une chaîne de production automobile. Il s'agit d'une adaptation du problème présenté dans le chapitre 3 au contexte des flux. L'ensemble des solutions de l'instance de cet exemple que nous utilisons est intégralement représentable sur papier sans pour autant être trop simpliste. Enfin, un dernier exemple, qui porte sur une adapation du Sudoku au contexte des flux, illustre un cas dans lequel la première méthode présentée dans le chapitre précédent est inapplicable, mais pour lequel la seconde méthode fonctionne très bien du fait d'une spécifité particulière du problème.

8.1 Feux de signalisation

Dans cet exemple, nous considérons un carrefour doté de deux voies de circulation qui se croisent. Afin de rendre la circulation plus sûre, le carrefour est équipé d'un système de signalisation consituté de quatre feux tricolores. Ces feux permettent d'autoriser ou de bloquer la circulation sur l'une ou l'autre des voies. La figure 8.1 représente le carrefour considéré.

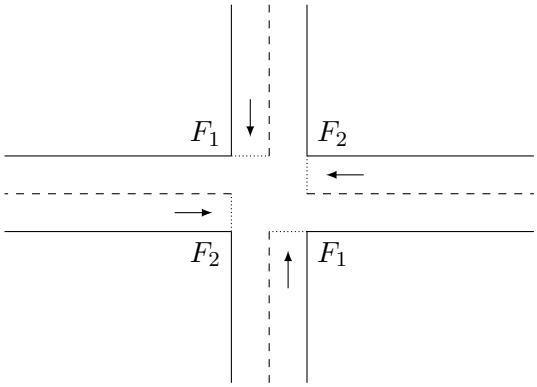

FIGURE 8.1 – Schéma du carrefour considéré dans l'exemple des feux de circulation

C_C	
V	R
O	R
R	V
R	O
R	R

C_E	
V	O
O	R
R	V
R	R

FIGURE 8.2 – Tables des contraintes C_C et C_E de l'exemple des feux de signalisation

8.1.1 Modélisation

Le problème est constitué de deux variables flux F_1 et F_2 de domaines $D(X) = D(Y) = \{V, O, R\}$. Les symboles V, O et R correspondent aux couleurs usuelles des feux tricolores, c'est-à-dire respectivement "vert", "orange" et "rouge". L'objectif du problème est de déterminer l'ensemble des séquences qui permettent d'assurer la sécurité de la circulation.

La circulation ne doit pas être autorisée sur les deux voies en même temps. Ceci est modélisé par la contrainte extensionnelle C_C, qui spécifie qu'au moins l'une des voies doit être fermée à tout instant, c'est-à-dire être dans l'état 'R'.

L'évolution des couleurs de chaque voie doit être prévisible. Le vert doit être suivi de l'orange, qui doit être suivi du rouge, qui doit être suivi du vert. Nous ajoutons également la possibilité que le rouge puisse être suivi du rouge, ce qui n'est pas strictement nécessaire pour une carrefour à deux feux, mais devient nécessaire à partir de trois feux. Ceci est modélisé par la contrainte extensionnelle C_E.

Les tables correspondant aux contraintes C_C et C_E sont données dans la figure 8.2, et les contraintes du problèmes sont données dans la figure 8.3.

$C_C(F_1, F_2)$	Contrainte de compatibilité
$C_E(F_1, \mathbf{X}F_1)$	Contrainte d'évolution de F_1
$C_E(F_2, \mathbf{X}F_2)$	Contrainte d'évolution de F_2

FIGURE 8.3 – Contrainte du problème sur les feux de signalisation

$C_C(F_1, F_2)$	Contrainte de compatibilité
$C_C(\mathbf{X}F_1, \mathbf{X}F_2)$	Contrainte de compatibilité à l'instant suivant
$C_E(F_1, \mathbf{X}F_1)$	Contrainte d'évolution de F_1
$C_E(F_2, \mathbf{X}F_2)$	Contrainte d'évolution de F_2

FIGURE 8.4 – Contraintes du CSP associé au problème des feux de signalisation

F_1	F_2	$\mathbf{X}F_1$	$\mathbf{X}F_2$
V	R	O	R
O	R	R	V
O	R	R	R
R	V	R	O
R	O	V	R
R	O	R	R
R	R	V	R
R	R	R	V
R	R	R	R

FIGURE 8.5 – Solutions du CSP associé au problème des feux de signalisation

8.1.2 Résolution

Pour la résolution de ce problème, l'étape de réduction de temporalité n'est pas nécessaire car le problème est déjà de temporalité 1. Le CSP permettant de calculer le système de transitions associé est donc une transcription directe des contraintes mentionnées dans la figures 8.3. Soient F_1 et F_2 les variables CSP à l'instant courant, et $\mathbf{X}F_1$ et $\mathbf{X}F_2$ les variables correspondantes à l'instant suivant, avec $\mathcal{D}(f) = \{V, O, R\}$ pour $f \in \{F_1, F_2, \mathbf{X}F_1, \mathbf{X}F_2\}$. Les contraintes du CSP correspondant sont données dans la figure 8.4, et l'ensemble des solutions est donné dans la figure 8.5.

Le graphe du système de transitions correspondant est donné dans la figure 8.6. Par exemple, la solution $\{F_1 \mapsto V, F_2 \mapsto R, \mathbf{X}F_1 \mapsto O, \mathbf{X}F_2 \mapsto R\}$ correspond à la transition $VR \to OR$ dans la figure, où l'origine de la transition correspond aux valeurs des termes de temporalité 0 et la destination aux valeurs des termes de temporalité 1.

Dans le formalisme des StCSPs, nous ne distinguons pas la notion d'état initial et d'état final, car nous cherchons simplement des séquences infinies qui satisfont des contraintes locales. Tout état qui permet de générer une solution infinie peut servir d'état initial dans ce contexte,

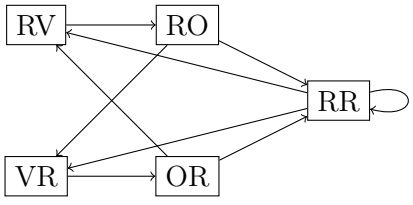

FIGURE 8.6 – Système de transitions du problème des feux de signalisation

$$\left\langle \begin{array}{c} F_1 \\ F_2 \end{array} \right\rangle = \left\langle \begin{array}{cccc} R & R & V & O \\ V & O & R & R \end{array} \right\rangle^{\omega}$$

FIGURE 8.7 – Exemple de solution périodique pour le problème des feux de signalisation

$$\left\langle \begin{array}{c} F_1 \\ F_2 \end{array} \right\rangle = \left[\left\langle \begin{array}{cccc} R & R & V & O \\ V & O & R & R \end{array} \right\rangle \left\langle \begin{array}{c} R \\ R \end{array} \right\rangle^{k} \right]_{k=1}^{\infty}$$

FIGURE 8.8 – Exemple de solution non-ultimement périodique pour le problème des feux de signalisation

et tous les états sont considérés finaux au sens de Büchi. L'exemple que nous traitons ici est très particulier, car sont système de transition forme une unique composante fortement connexe, ce qui permet de passer de n'importe quel état à n'importe quel autre. Une solution périodique est donnée dans la figure 8.7.

Comme nous l'avons mentionné dans le chapitre précédent, la représentation des solutions non ultimement périodiques est beaucoup moins facile. Nous donnons un exemple de solution non-ultimement périodique dans la figure 8.8, en utilisant un opérateur de concaténation généralisé $[w(k)]_{k=k_{\min}}^{k_{\max}}$ pour dénoter le mot $w(k_{\min}) \cdot w(k_{\min}+1) \cdots w(k_{\max})$ si $k_{\max} \in \mathbb{N}$ et $w(k_{\min}) \cdot w(k_{\min}+1) \cdots$ si $k_{\max} = \infty$.

8.1.3 Discussion

Le problème de feux de circulation décrit ici est volontairement extrêment simpliste. Le champ de la gestion du traffic urbain est une discipline complexe qui utilise des techniques extrêment diverses.

Parmi les extensions naturelles qui ne sont pas prises en compte dans le modèle présenté ici, nous pouvons considérer désirable de faire en sorte que chaque feu de signalisation finisse toujours par passer au vert, et que la durée d'un feu rouge ne dure pas trop longtemps. Les différences de durées entre les phases rouges, oranges et vertes peuvent être facilement modélisée par des contraintes de temporalité plus grande, mais cela nécessite d'utiliser des types de contraintes particulières, car une définition extensionnelle serait difficile à manipuler. L'état dans lequel toutes les voies sont au rouge n'est pas nécessaire pour un carrefour à deux feux, mais devient nécessaire dans un problème à plus de deux feux, car la durée d'un passage au rouge doit être suffisamment longue pour que toutes les autres voies puissent passer. La prise en compte de feux pour les passages piétons induit nécessaire d'autre contraintes. Il est également possible d'utiliser des feux différents suivant que les usagers tournent à gauche ou continunent tout droit. Cette distinction induit la nécessité de gérer les intersection de voies afin de produire des contraintes de compatibilité adaptées.

L'optimisation du traffic nécessite de spécifier des hypothèses sur le traffic escompté, afin de faire en sorte que les voies les plus chargées circulent plus fluidement que les voies moins chargée. Les méthodes habituelles utilisent souvent différentes hypothèses suivant les jours de la semaines et selon les heures, pour lesquelles les solutions optimales sont différentes, et les plages horaires de ces conditions sont codées directement dans le contrôleur.

C_C				
1	0	1	1	0
0	0	0	1	0
0	1	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	0

TABLE 8.1 – Table de la contrainte C_C

L'optimisation de réseaux est beaucoup plus difficile. En particulier, la génération automatisée de "vague verte", dans lesquelles l'usager qui part à une intersection à un feu vert arrive à la prochaine destination à un feu également vert, nécessite d'établir le temps moyen nécessaire pour le trajet de l'usager entre les deux intersections, ainsi que le traffic escompté pour chaque voies qui mènent à un feu de destination. Comme chaque feu peut recevoir du traffic émanant de plusieurs voie, le problème du contrôle optimal d'un tel système est assez éloigné du cadre des StCSP, car il s'agit alors d'un problème de contrôle probabiliste.

8.2 Problème d'ordonnancement d'une chaîne de production automobile

Dans cet section, nous considérons un exemple d'application des StCSPs au problème de l'ordonnancement d'une chaîne de prodcution automobile. Il s'agit d'une adaptation du problème [127] au contexte des flux, lequel a été présenté dans le section 3.6. Nous utilisons ici les mêmes options, classes et contraintes de capacité.

Dans le contexte des flux, l'objectif naturel est de calculer des séquences de classes compatibles avec les contraintes de la chaîne de production. Ces séquences sont infinies, ce qui rend les contraintes sur les nombres exacts de classes de chaque type diffiles à adapter au cas des variables flux. Nous examinons la possibilité d'intégrer ce type de contrainte quantitative à la fin de la section.

8.2.1 Modélisation

La modélisation est similaire dans l'idée à celle décrite dans la section 3.6. Elle consiste en cinq variables flux booléennes, où chaque variable correspond à une option. Les classes sont représentées par une contrainte extensionnelle C_C de temporalité 0, qui restreint les combinaisons d'options possibles à chaque instant à l'une des classes autorisées. Les contraintes de capacités sont toujours représentées par des contraintes linéaires, la différence étant qu'elles portent ici sur des variables flux.

Les variables sont $\mathcal{X} = \{O_1, O_2, O_3, O_4, O_5\}$, de domaines $D(O_i) = \mathbb{B}^\omega$ et donc de base $\Sigma(O_i) = \mathbb{B}$. Les contraintes sont données dans la table 8.2. Le détail de la contrainte C_C est donné dans la table 8.1.

Ce problème est de temporalité 4. Il est donc nécessaire de le transformer en un StCSP équivalent de temporalité 1 afin de pouvoir construire le système de transitions associé. Nous

Contrainte de classe	$C_C(O_1, O_2, O_3, O_4, O_5)$
Capacité de l'option O_1	$O_1 + \mathbf{X}O_1 \leqslant 1$
Capacité de l'option O_2	$O_2 + \mathbf{X}O_2 + \mathbf{X}^2O_2 \leqslant 2$
Capacité de l'option O_3	$O_3 + \mathbf{X}O_3 + \mathbf{X}^2O_3 \leqslant 1$
Capacité de l'option O_4	$O_4 + \mathbf{X}O_4 + \mathbf{X}^2O_4 + \mathbf{X}^3O_4 + \mathbf{X}^4O_4 \leqslant 2$
Capacité de l'option O_5	$O_5 + \mathbf{X}O_5 + \mathbf{X}^2O_5 + \mathbf{X}^3O_5 + \mathbf{X}^4O_5 \leqslant 1$

TABLE 8.2 – Contraintes du problème d'ordonnancement automobile

Contraintes	
Contrainte de classe	$C_C(O_1, O_2, O_3, O_4, O_5)$
Capacité de l'option O_1	$O_1 + \mathbf{X}O_1 \leqslant 1$
Capacité de l'option O_2	$O_2 + O_2^1 + \mathbf{X}O_2^1 \leqslant 2$
Capacité de l'option O_3	$O_3 + O_3^1 + \mathbf{X}O_3^1 \leqslant 1$
Capacité de l'option O_4	$O_4 + O_4^1 + O_4^2 + O_4^3 + \mathbf{X}O_4^3 \leqslant 2$
Capacité de l'option O_5	$O_5 + O_5^1 + O_5^2 + O_5^3 + \mathbf{X}O_5^3 \leqslant 1$
Contraintes d'égalité	
Contraint d'égalité pour O_2^1	$O_2^1 = \mathbf{X}O_2$
Contraint d'égalité pour O_3^1	$O_3^1 = \mathbf{X}O_3$
Contraint d'égalité pour O_4^1	$O_4^1 = \mathbf{X}O_4$
Contraint d'égalité pour O_4^2	$O_4^2 = \mathbf{X}O_4^1$
Contraint d'égalité pour O_4^3	$O_4^3 = \mathbf{X}O_4^2$
Contraint d'égalité pour O_5^1	$O_5^1 = \mathbf{X}O_5$
Contraint d'égalité pour O_5^2	$O_5^2 = \mathbf{X}O_5^1$
Contraint d'égalité pour O_5^3	$O_5^3 = \mathbf{X}O_5^2$

TABLE 8.3 – Contraintes du problème d'ordonnancement automobile réduit

dénotons les variables auxiliaires O_i^j, qui correspondent au terme $\mathbf{X}^jO_i$. Les contraintes du StCSP réduit sont données dans la table 8.3.

Le nombre de solutions du CSP associé est trop grand pour que celles-ci soient intégralemnt données ici. Le graphe du système de transitions associé est donné dans la figure 8.9. Les lettres A, B, C, D, E, F correspondent aux 6 classes du problèmes. Les variables auxiliaires O_i^j avec $j > 0$ sont nécessaires pour définir les états, mais peuvent être omises lors de la génération des solutions car, comme expliqué dans le chapitre 6, elles servent seulement à définir la structure de l'automate. C'est pourquoi ces variables sont omises dans la figure. L'utilisation d'étiquettes pour les classes permet de représenter concisément l'information utile, mais cette présentation est le résulat d'un post-traitement qui est spécifique au problème d'ordonnancement.

Dans la figure 8.9, les impasses ont été supprimées, car l'algorithme génère un grand nombre d'états qui mènent dans des impasses. Le graphe nettoyé est constitué d'une unique composante connexe. Il ne s'agit pas d'une propriété du problème d'ordonnancement, mais d'une propriété de l'instance.

La notion d'automate contient généralement la notion d'état initial et d'état final. Dans le contexte actuel des StCSPs, tous les états qui ne mènent pas à des impasses sont utilisables

comme états initiaux, et la notion d'état final n'est pas spécifiée ici. Conceptuellement, tous les états sont finaux.

8.2.2 Comparaison avec la version PPC du problème

Les version PPC et StCSP du problème d'ordonnancement d'une chaîne de production automobile sont très similaires du point de vue des contraintes. La seule contrainte manquante dans l'adaption aux StCSPs est la contrainte de cardinalité sur le nombre de classes.

Cependant, la version PPC ne permet d'obtenir qu'une quantité limitée d'information, alors que le système de transitions correspondant à la version StCSP est beaucoup plus riche.

Par exemple, nous pouvons observer que la classe F ne peut être présente qu'une seule fois en début de séquence. Il est donc impossible de construire une séquence qui contiendra plusieurs fois cette classe, et si cela est désiré, il faudra modifier les contraintes de capacité correspondantes.

En supprimant la classe F, le système de transitions devient fortement connexe, ce qui permet d'avoir la garantie qu'il est toujours possible de générer une voiture du modèle voulu quelque soit l'état du sytème. Cependant, plusieurs chemins sont possibles pour arriver à la génération d'un modèle demandé, et impliquent généralement la construction de modèles non demandés. Dans un tel contexte, le choix d'un chemin particulier devient un problème de contrôle.

8.2.3 Adapation de la contrainte de cardinalité

La difficulté du problème [127] provient de la contrainte de cardinalité sur les nombres exacts de véhicules désirés de chaque classe. Il s'agit d'un contrainte globale très forte qui s'ajoute aux contraintes locales sur les contraintes de classe et de transition.

Le formalisme des StCSP ne permet absolument pas de simplifier la résolution du problème original, car une fois le calcul du système de transitions effectué, la seule possibilité serait d'examiner le graphe en utilisant un parcours en profondeur sur un horizon fini. Ce parcours pourrait s'arrêter dès que le nombre de véhicules produit pour un modèle donné devient trop grand, mais le nombre de chemins considérés serait probablement équivalent au nombre de choix considérés par un algorithme de satisfaction de contraintes classique.

Dans le cas des flux infinis, il existe toujours au moins une classe d'automobiles qui est produite un nombre infini de fois. Il est donc impossible d'imposer une contrainte de cardinalité aussi stricte. En revanche, en utilisant les résultats de [28], il est possible d'imposer des contraintes sur le nombre moyen de voitures produites. Il suffit pour cela d'étiqueter chaque transition par une fonction qui ajoute une unité à la composante correspondant à la classe de destination, et qui laisse les autres classes invariantes. Il est alors possible de déterminer des trajectoires qui permettent d'assurer que les nombres moyens restent dans des limites désirées, si cela est possible.

En terme de contrainte, ce type de critère doit être appliqué sur l'intégralité du système de transitions, ce qui est possible seulement s'il est de taille raisonnable. Il s'agit d'un problème de contrôle, et la résolution de ce type de problème nécessite des techniques différentes qui prennent le système de transitions comme paramètre.

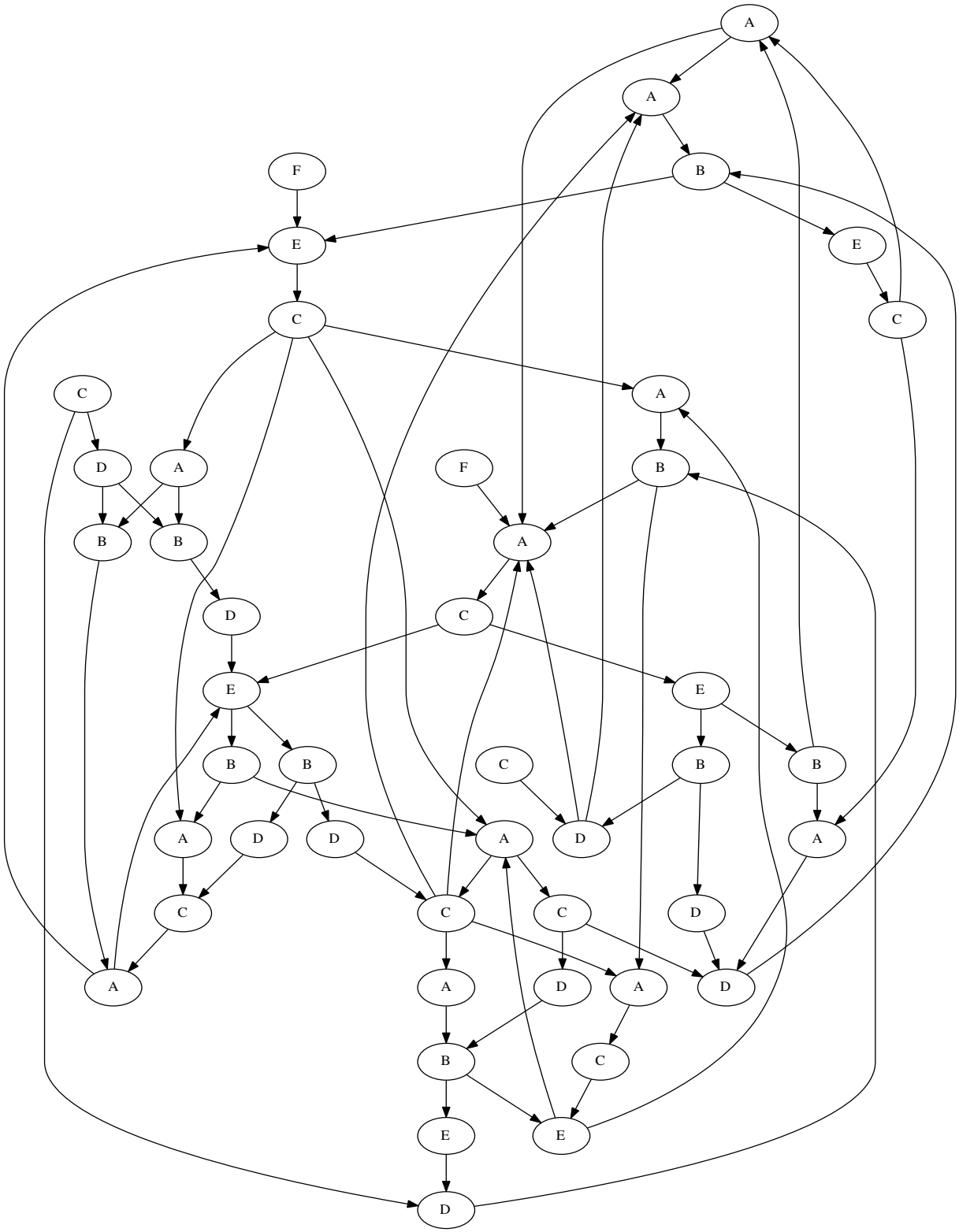

FIGURE 8.9 – Système de transitions associé au problème d'ordonnancement de production automobile

Colonnes
$\texttt{alldiff}(C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9)$
Lignes
$\texttt{alldiff}(C_1, \mathbf{X}^1C_1, \mathbf{X}^2C_1, \mathbf{X}^3C_1, \mathbf{X}^4C_1, \mathbf{X}^5C_1, \mathbf{X}^6C_1, \mathbf{X}^7C_1, \mathbf{X}^8C_1)$
$\texttt{alldiff}(C_2, \mathbf{X}^1C_2, \mathbf{X}^2C_2, \mathbf{X}^3C_2, \mathbf{X}^4C_2, \mathbf{X}^5C_2, \mathbf{X}^6C_2, \mathbf{X}^7C_2, \mathbf{X}^8C_2)$
$\texttt{alldiff}(C_3, \mathbf{X}^1C_3, \mathbf{X}^2C_3, \mathbf{X}^3C_3, \mathbf{X}^4C_3, \mathbf{X}^5C_3, \mathbf{X}^6C_3, \mathbf{X}^7C_3, \mathbf{X}^8C_3)$
$\texttt{alldiff}(C_4, \mathbf{X}^1C_4, \mathbf{X}^2C_4, \mathbf{X}^3C_4, \mathbf{X}^4C_4, \mathbf{X}^5C_4, \mathbf{X}^6C_4, \mathbf{X}^7C_4, \mathbf{X}^8C_4)$
$\texttt{alldiff}(C_5, \mathbf{X}^1C_5, \mathbf{X}^2C_5, \mathbf{X}^3C_5, \mathbf{X}^4C_5, \mathbf{X}^5C_5, \mathbf{X}^6C_5, \mathbf{X}^7C_5, \mathbf{X}^8C_5)$
$\texttt{alldiff}(C_6, \mathbf{X}^1C_6, \mathbf{X}^2C_6, \mathbf{X}^3C_6, \mathbf{X}^4C_6, \mathbf{X}^5C_6, \mathbf{X}^6C_6, \mathbf{X}^7C_6, \mathbf{X}^8C_6)$
$\texttt{alldiff}(C_7, \mathbf{X}^1C_7, \mathbf{X}^2C_7, \mathbf{X}^3C_7, \mathbf{X}^4C_7, \mathbf{X}^5C_7, \mathbf{X}^6C_7, \mathbf{X}^7C_7, \mathbf{X}^8C_7)$
$\texttt{alldiff}(C_8, \mathbf{X}^1C_8, \mathbf{X}^2C_8, \mathbf{X}^3C_8, \mathbf{X}^4C_8, \mathbf{X}^5C_8, \mathbf{X}^6C_8, \mathbf{X}^7C_8, \mathbf{X}^8C_8)$
$\texttt{alldiff}(C_9, \mathbf{X}^1C_9, \mathbf{X}^2C_9, \mathbf{X}^3C_9, \mathbf{X}^4C_9, \mathbf{X}^5C_9, \mathbf{X}^6C_9, \mathbf{X}^7C_9, \mathbf{X}^8C_9)$
Blocs
$\texttt{alldiff}(C_1, \mathbf{X}^1C_1, \mathbf{X}^2C_1, C_2, \mathbf{X}^1C_2, \mathbf{X}^2C_2, C_3, \mathbf{X}^1C_3, \mathbf{X}^2C_3)$
$\texttt{alldiff}(C_4, \mathbf{X}^1C_4, \mathbf{X}^2C_4, C_5, \mathbf{X}^1C_5, \mathbf{X}^2C_5, C_6, \mathbf{X}^1C_6, \mathbf{X}^2C_6)$
$\texttt{alldiff}(C_7, \mathbf{X}^1C_7, \mathbf{X}^2C_7, C_8, \mathbf{X}^1C_8, \mathbf{X}^2C_8, C_9, \mathbf{X}^1C_9, \mathbf{X}^2C_9)$
$\texttt{alldiff}(\mathbf{X}^3C_1, \mathbf{X}^4C_1, \mathbf{X}^5C_1, \mathbf{X}^3C_2, \mathbf{X}^4C_2, \mathbf{X}^5C_2, \mathbf{X}^3C_3, \mathbf{X}^4C_3, \mathbf{X}^5C_3)$
$\texttt{alldiff}(\mathbf{X}^3C_4, \mathbf{X}^4C_4, \mathbf{X}^5C_4, \mathbf{X}^3C_5, \mathbf{X}^4C_5, \mathbf{X}^5C_5, \mathbf{X}^3C_6, \mathbf{X}^4C_6, \mathbf{X}^5C_6)$
$\texttt{alldiff}(\mathbf{X}^3C_7, \mathbf{X}^4C_7, \mathbf{X}^5C_7, \mathbf{X}^3C_8, \mathbf{X}^4C_8, \mathbf{X}^5C_8, \mathbf{X}^3C_9, \mathbf{X}^4C_9, \mathbf{X}^5C_9)$
$\texttt{alldiff}(\mathbf{X}^6C_1, \mathbf{X}^7C_1, \mathbf{X}^8C_1, \mathbf{X}^6C_2, \mathbf{X}^7C_2, \mathbf{X}^8C_2, \mathbf{X}^6C_3, \mathbf{X}^7C_3, \mathbf{X}^8C_3)$
$\texttt{alldiff}(\mathbf{X}^6C_4, \mathbf{X}^7C_4, \mathbf{X}^8C_4, \mathbf{X}^6C_5, \mathbf{X}^7C_5, \mathbf{X}^8C_5, \mathbf{X}^6C_6, \mathbf{X}^7C_6, \mathbf{X}^8C_6)$
$\texttt{alldiff}(\mathbf{X}^6C_7, \mathbf{X}^7C_7, \mathbf{X}^8C_7, \mathbf{X}^6C_8, \mathbf{X}^7C_8, \mathbf{X}^8C_8, \mathbf{X}^6C_9, \mathbf{X}^7C_9, \mathbf{X}^8C_9)$

TABLE 8.4 – Contraintes du Sudoku glissant

8.3 Sudoku glissant

Le Sudoku est un puzzle numérique d'origine nippone basé sur la contrainte globale "tous différents". Il se joue sur une grille de 9×9 cases, dont les domaines sont les chiffres de 1 à 9. Une grille est une solution si les chiffres sur chaque ligne, sur chaque colonne, et sur chaque bloc sont tous différents les uns des autres. Les blocs sont les neuf sous-grilles de 3×3 cases, et ces grilles sont agencées de telle sorte qu'aucune case ne fasse partie de deux sous-grilles à la fois. Autrement dit, les neufs blocs forment eux-mêmes une grille 3×3 blocs.

Ce jeu est un problème intéressant de programmation par contraintes, car il s'exprime facilement, contient un très grand nombre de solutions, et contient beaucoup de symétries. En particulier, le grand nombre de solutions fait que l'énumération de toutes les solutions est en pratique impossible, et le calcul du nombre de solutions doit se faire de façons analytique [50, 74]. Une présentation des Sudoku en programmation par contraintes figure dans l'introduction de [131], ainsi que dans [33] et [122].

Nous présentons ici une adaptation du Sudoku aux StCSPs, que nous baptisons "Sudoku glissant". L'idée consiste à considérer une grille de longueur infinie de 9 cases de haut, et d'imposer que toutes les grilles de 9×9 cases le long de la séquence sont des solutions du Sudoku.

La modélisation est constituée de neuf variables flux $\mathcal{X} = \{C_1, \ldots, C_9\}$ de domaines $\mathcal{D}(C_i) = \{1, \ldots, 9\}$. Les contraintes sont données dans la table 8.4.

$$\begin{Bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \\ C_6 \\ C_7 \\ C_8 \\ C_9 \end{Bmatrix} = \begin{Bmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \\ 8 & 9 \\ 2 & 3 \\ 5 & 6 \\ 6 & 7 \\ 9 & 1 \\ 3 & 4 \end{Bmatrix} \cdot \begin{Bmatrix} 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 \\ 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 \\ 9 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 \\ 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 & 6 \\ 8 & 9 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 \\ 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 \end{Bmatrix}^{\omega}$$

FIGURE 8.10 – Une solution du Sudoku glissant

La sémantique des contraintes de voisinage fait que toutes les sudoku ne sont pas nécessairement extensibles à des Sudoku glissants, car les contraintes de blocs doivent être toujours satisfaites lorsqu'elles sont décalées. Une solution du Sudoku glissant est donnée dans la figure 8.10. Cete solution est présentée de telle sorte qu'il soit facile de vérifier que toutes les contraintes sont satisfaites.

La première méthode est inappliquable au problème du Sudoku glissant, car le système de transitions est extrêment grand. Ceci rend nécessaire la navigation dans l'espace de recherche. En revanche, les propriétés du problème font que les solutions sont nécessairement périodiques, car une fois une solution trouvée pour les 11 premiers instants, il existe une seule manière de continuer la séquence. L'algorithme de parcours en profondeur doit donc seulement trouver 3 transitions correctes pour obtenir un cycle, ce qui le rend particulièrement efficace dans ce cas.

Chapitre 9

StCSPs et logiques temporelles

Sommaire

Dans ce chapitre, nous étudions les relations entre les StCSPs et les logiques temporelles. Ces comparaisons sont importantes, car elles permettent d'intégrer les nombreux résultats qui concernent les logiques temporelles dans le contexte de la satisfaction de contraintes. En particulier, les algorithmes de satisfaction et de vérification, ainsi que leurs complexités, constituent une ressource pour la mise en œuvre d'algorithmes de satisfaction de contraintes sur les flux.

Les résultats présentés dans ce chapitre s'appuient principalement sur le contenu du chapitre 4. Dans la section 9.1, nous décrivons une équivalence exponentielle entre les StCSPs et un fragment de la logique temporelle propositionnelle linéaire, par le biais d'une procédure qui permet de transformer une instance d'un StCSP en une formule LTL équivalente. Nous discutons ensuite dans la section 9.2 de la possibilité d'augmenter l'expressivité des StCSPs, et nous concluons sur les avantages et inconvénients des StCSPs comparés à d'autres formalismes pour l'expression et la génération d'ensembles de séquences infinies.

9.1 Transformation d'un StCSP en spécification LTL

Nous décrivons dans cette section une procédure de transformation d'une instance d'un StCSP en une formule de logique LTL. La transformation décrite est similaire à celle qui est utilisée pour encoder un CSP en problème de satisfaction booléen, et le gain en concision obtenu par les StCSP est similaire à celui obtenu par le formalisme de la programmation par contraintes. Ceci implique que le formalisme des StCSPs n'est pas plus expressif que la logique LTL, mais est plus concis que le fragment correspondant de cette logique. Les notions d'expressivité et de concision pour les logiques temporelles sont détaillées dans [89].

Soit $\mathcal{S} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ un StCSP comportant uniquement des contraintes de voisinage, constitué de n variables $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$, de bases $\Sigma(x_i) = \{v_{i,1}, \ldots, v_{i,m_i}\}$, où m_i est la taille de $\Sigma(x_i)$ pour $i \in \{1, \ldots, n\}$.

Afin de constuire une formule LTL équivalente à $\mathcal{S}$, nous introduisons l'ensemble de variables propositionnelles $\text{Prop}(\mathcal{S})$, dans lequel chaque variable code une valeur d'une variable flux :

$$\text{Prop}(\mathcal{S}) = \{p_{i,j} \mid x_i \in \mathcal{X} \wedge v_{i,j} \in \Sigma(x_i)\}$$

Le nombre de variables propositionnelles est la somme de la taille des bases des variables :

$$|\text{Prop}(\mathcal{S})| = \sum_{1 \leqslant i \leqslant n} |\Sigma(x_i)|$$

La famille de formules propositionnelles $\{\varphi_{x_i}\}_{1 \leqslant i \leqslant k}$ suivante permet d'assurer qu'une variable ne peut prendre qu'une valeur de son domaine à un instant donné.

$$\varphi_{x_i} = \bigvee_{1 \leqslant j \leqslant m_i} \left(p_{i,j} \wedge \bigwedge_{\substack{1 \leqslant j' \leqslant m_i \\ j' \neq j}} \neg p_{i,j'} \right)$$

La conjonction de ces formules permet d'assurer que la sémantique de valeur unique est respectée pour toutes les variables :

$$\varphi_{\mathcal{X}} = \bigwedge_{1 \leqslant i \leqslant n} \varphi_{x_i}$$

La taille de $\varphi_{\mathcal{X}}$ est donnée par :

$$|\varphi_{\mathcal{X}}| = \sum_{1 \leqslant i \leqslant n} m_i^2$$

L'intégration des contraintes de voisinage en logique temporelle est similaire à celle effectuée pour obtenir un encodage en satisfaction propositionnelle. Soit $c(\mathbf{X}^{t_1} x_{r_1}, \dots \mathbf{X}^{t_k} x_{r_k}) \in \mathcal{C}$ une contrainte de voisinage, et $\text{Sol}(c)$ l'ensemble des solutions de c interprétée comme une contrainte PPC sur l'ensemble de variables $\text{var}(c) = \{\mathbf{X}^{t_1} x_{r_1}, \dots \mathbf{X}^{t_k} x_{r_k}\}$ de domaines $\Sigma(x_{r_i})$ pour $1 \leqslant i \leqslant k$, où k, $\{r_i\}_{1 \leqslant i \leqslant k}$ et $\{t_i\}_{1 \leqslant i \leqslant k}$ sont dépendants de c.

Soit $\iota : \mathcal{X} \times \mathbb{Z} \to \text{Prop}(\mathcal{S})$ la fonction qui associe à une paire $\langle x, v \rangle$ telle que $v \in \Sigma(x)$ la variable propositionnelle $p_{x,v}$.

La formule propositionnelle temporelle équivalente à c est alors la suivante :

$$\varphi_c = \bigvee_{I \in \text{Sol}(c)} \left(\bigwedge_{1 \leqslant i \leqslant k} \mathbf{X}^{t_i} \iota(x_{r_i}, I(\mathbf{X}^{t_i} x_{r_i})) \right)$$

La taille de φ_c est linéaire en le nombre de solutions de c, c'est-à-dire de façon générale exponentielle en la taille des domaines.

La formule qui permet de spécifier toutes les contraintes est alors la suivante :

$$\varphi_{\mathcal{C}} = \bigwedge_{c \in \mathcal{C}} \varphi_c$$

et la formule LTL qui correspond à $\mathcal{S}$ est :

$$\varphi_{\mathcal{S}} = \mathbf{G}\left(\varphi_{\mathcal{X}} \wedge \varphi_{\mathcal{C}}\right)$$

$\varphi_{\mathcal{S}}$ est principalement constituée de l'encodage des contraintes, car l'encodage de la sémantique de valeur unique ne constitue qu'une partie quadratique de la formule, alors que l'encodage des contraintes est exponentielle. De plus, pour un StCSP $\mathcal{S}$, la formule LTL $\varphi_{\mathcal{S}}$ associée est nécessairement de la forme $\mathbf{G}\psi$ où ψ est un fragment équivalent à la logique propositionnelle lorsque l'on considère chaque terme de la forme $\mathbf{X}^k p$ comme une variable propostionnelle, avec $k \in \mathbb{N}$ et p une variable propositionnelle.

9.1.1 Exemple

Nous considérons l'exemple des feux de signalisation de la section 8.1.

Nous utilisons les variables propositionnelles $\mathrm{Prop}(\mathcal{S}) = \{p_{1\mathrm{V}}, p_{1\mathrm{O}}, p_{1\mathrm{R}}, p_{2\mathrm{V}}, p_{2\mathrm{O}}, p_{2\mathrm{R}}\}$.

Concernant les contraintes de domaines, les formules sont les suivantes :

$$\begin{aligned}\varphi_{F_1} &= (p_{2\mathrm{V}} \wedge \neg p_{2\mathrm{O}} \wedge \neg p_{2\mathrm{R}}) \\ &\vee (\neg p_{2\mathrm{V}} \wedge p_{2\mathrm{O}} \wedge \neg p_{2\mathrm{R}}) \\ &\vee (\neg p_{2\mathrm{V}} \wedge \neg p_{2\mathrm{O}} \wedge p_{2\mathrm{R}})\end{aligned}$$

$$\begin{aligned}\varphi_{F_2} &= (p_{1\mathrm{V}} \wedge \neg p_{1\mathrm{O}} \wedge \neg p_{1\mathrm{R}}) \\ &\vee (\neg p_{1\mathrm{V}} \wedge p_{1\mathrm{O}} \wedge \neg p_{1\mathrm{R}}) \\ &\vee (\neg p_{1\mathrm{V}} \wedge \neg p_{1\mathrm{O}} \wedge p_{1\mathrm{R}})\end{aligned}$$

La contrainte de comptabilité est la suivante :

$$\begin{aligned}\varphi_{C_C} &= (p_{1\mathrm{V}} \wedge p_{2\mathrm{O}}) \\ &\vee (p_{1\mathrm{O}} \wedge p_{2\mathrm{R}}) \\ &\vee (p_{1\mathrm{R}} \wedge p_{2\mathrm{V}}) \\ &\vee (p_{1\mathrm{R}} \wedge p_{2\mathrm{O}}) \\ &\vee (p_{1\mathrm{R}} \wedge p_{2\mathrm{R}})\end{aligned}$$

Et les contraintes d'évolution se calculent de façon similaire :

$$\begin{aligned}\varphi_{C_E,F_1} &= (p_{1\mathrm{V}} \wedge \mathbf{X}p_{1\mathrm{O}}) \\ &\vee (p_{1\mathrm{O}} \wedge \mathbf{X}p_{1\mathrm{R}}) \\ &\vee (p_{1\mathrm{R}} \wedge \mathbf{X}p_{1\mathrm{V}}) \\ &\vee (p_{1\mathrm{R}} \wedge \mathbf{X}p_{1\mathrm{R}})\end{aligned}$$

$$\begin{aligned}\varphi_{C_E,F_2} &= (p_{2\mathrm{V}} \wedge \mathbf{X}p_{2\mathrm{O}}) \\ &\vee (p_{2\mathrm{O}} \wedge \mathbf{X}p_{2\mathrm{R}}) \\ &\vee (p_{2\mathrm{R}} \wedge \mathbf{X}p_{2\mathrm{V}}) \\ &\vee (p_{2\mathrm{R}} \wedge \mathbf{X}p_{2\mathrm{R}})\end{aligned}$$

La formule complète est donnée par le développement de la formule :

$$\varphi = \varphi_{F_1} \wedge \varphi_{F_2} \wedge \varphi_{C_C} \wedge \varphi_{C_E,F_1} \wedge \varphi_{C_E,F_2}$$

L'automate de Büchi calculé sur cette formule par la procédure décrite dans le chapitre 4 produit un système de transitions similaire à celui donné dans la section 8.1. La notion d'état final nécessaire pour les formules LTL n'est pas nécessaire pour les StCSPs, car les opérateurs **U** et **F** ne sont pas utilisés.

9.2 Pistes pour la généralisation

L'exemple de la section précédente montre que notre formalisme offre un avantage expressif par rapport aux formalisme des logiques temporelles. La transformation décrite dans cette section est optimale, dans le sens où l'expansion des contraintes en formules de logique propositionnelle est nécessairement exponentielle dans le cas général.

En terme de complexité, nous rappelons que le problème de satisfaction de contraintes est NP-complet, alors que le problème de satisfaction de la logique temporelle linéaire est PSPACE-complet. La logique LTL restreint à la modalité '**X**' est équivalent à la logique propositionnelle, donc son problème de satisfaction est NP-complet. L'inclusion de l'opérateur '**G**' induit la nécessité de gérer la notion de clotûre transitive [62], qui nécessite de travailler sur le système de transitions.

La complexité de la résolution des StCSPs correspond au gain en concision apporté par l'utilisation des contraintes. Les méthodes de résolution que nous proposons sont équivalentes en terme de complexité à ce qui serait obtenu en utilisant une traduction directe en logique LTL, mais nos méthodes permettent de maximiser l'utilisation des algorithmes de propagation, ce qui permet d'exploiter la sémantique des contraintes, laquelle serait perdue en utilisant un passage vers la logique LTL.

Le fragment que nous utilisons pour l'encodage est strictement moins expressif que la logique LTL. Soit $\varphi = \mathbf{G}\psi$ la formule LTL associée à un StCSP. L'ajout d'une formule $\mathbf{F}\theta$ telle que $\varphi = \mathbf{G}\psi \wedge \mathbf{F}\theta$ ne poserait pas de problème particulier, car il suffirait de chercher les états du systèmes de transitions dans lesquels la formule θ est satisfaite. Ce type de contrainte ne ferait que compliquer légèrement l'algorithme présenté dans le chapitre 7.

L'extension du formalisme des StCSPs à la logique LTL complète nécessiterait de pouvoir calculer l'automate de Büchi associé à une formule LTL par énumération des solutions d'un CSP. Cependant, l'algorithme présenté dans 4 est récursif, sa condition d'arrêt étant qu'aucun états supplémentaires ne peut être généré. L'expression des transitions de l'automate par un CSP pose des problèmes de modélisation.

Soit $\varphi = \psi\mathbf{U}\theta$ une formule LTL. Alors les contraintes seraient intuitivement de la forme $p_\varphi \wedge p_\theta$ et $p_\varphi \wedge p_\psi \wedge \mathbf{X}p_\varphi$, en utilisant des variables propositionnelles p_ξ pour dénoter toutes les formules ξ qui sont utilisés dans les ensembles de formules de l'algorithme du chapitre 4, et en utilisant des termes de la forme $\mathbf{X}p$ pour dénoter les valeurs de ces variables à l'instant suivant. Même en considérant ψ et θ comme des variables propositionnelles, les solutions de ce CSP sont données dans la figure 9.1 illustrent le problème, qui est que les variables qui ne sont pas contraintes peuvent prendre n'importe quelles valeurs.

Le résultat désiré dans le cas présent aurait toutes les variables non contraintes positionnées à $\bot$, et il est difficile d'imposer cela par des contraintes locales, car rien ne dit que φ ne sera pas "membre" d'un ensemble plus grand contenant aussi ψ par exemple. Par ailleurs, la formule ϕ peut être une sous formule d'une formule plus grande. Dans ce cas, il existe d'autres variables inconnues de la contraintes qui doivent toutes être positionnées à $\bot$.

φ	ψ	θ	$\mathbf{X}\varphi$	$\mathbf{X}\psi$	$\mathbf{X}\theta$
$\top$	$*$	$\top$	$*$	$*$	$*$
$\top$	$\top$	$*$	$\top$	$*$	$*$

FIGURE 9.1 – Solutions du CSP associé à $\varphi = \psi \mathbf{U} \theta$

S'il n'est pas possible de construire l'automate de Büchi associé à un StCSP par énumération des solutions d'un StCSPs, alors il n'est pas possible de résoudre des StCSPs généralisés à la logique LTL de cette façon, est d'autres méthodes doivent être explorées. Une adaption de l'algorithme du chapitre 4 intégrant la PPC à un niveau plus basique semble plus réalisable, mais le seul apport d'une telle méthode serait un gain en concision.

Chapitre 10

Conclusion

Nous avons entrepris d'étudier dans ce manuscrit l'application des techniques de la programmation par contraintes au contexte des flux infinis. Nous avons pour cela choisi de créer un nouveau cadre simple, celui des StCSPs, et de le mettre en relation avec les formalismes existants.

Le nombre de travaux sur les séquences infinies est assez large et diversifié. Les formalismes les plus naturels sont ceux des logiques temporelles, mais le nombre de logiques temporelles lui-même est très large, et les méthodes algorithmiques qui les concernent sont également nombreuses et complexes. La mise en relation des techniques de satisfaction de contraintes et en particulier des techniques de propagation avec ces formalismes est par conséquent un travail difficile.

Nous pensons que la logique idéale pour la problématique de la satisfaction de contraintes sur les flux est la logique νTL. Cette logique est la restriction de la logique μTL, qui est la logique temporelle la plus générale qui soit, car elle englobe la plupart des autres logiques temporelles communément étudiées. Cependant, le travail présenté ici n'est équivalent qu'à un fragment de la logique LTL. L'extension des StCSP à la logique LTL en gardant les mêmes principes algorithmique doit être prouvée, mais il est évident que le problème de satisfaction pour les StCSPs, même étendus aux logiques à point-fixe, est décidable. La calsse des langages ω-réguleir semble incontournable, car très peu de travaux sur les mots infinis portent sur des classes de langages différentes qui ne sont pas des sous-ensembles de celle-ci. Certains travaux portent sur la généralisation des classes de langages de la hiérarchie de Chomsky aux mots infinis [48], et aux arbres infinis [102], mais les travaux qui portent sur des classes de langages distincts mais ne contenant pas les langages ω-réguliers où n'étant pas contenu dans ceux-ci sont beaucoup plus rares.

Au vu de la multitude de formalismes qui ont été produits dans les cinquantes dernières années, la comparaision et l'unification de ceux-ci est un thème fréquent. Ce thème est maintenant classie pour la programmation par contraintes, le problème de satisfaction de la logique propositionnelle, la planification et les problèmes d'ordonnancement. En ce qui concerne les logiques temporelles, les relations que nous avons exposées dans le chapitre 4 invitent à reconsidérer les algorithmes existants comme des adaptations des algorithmes qui s'appliquent aux formalismes les plus expressifs.

Certaines relations sont un peu moins connues, notamment celles entre *model checking* et planification, par exemple. Les formalismes temporels différents que ces domaiens utilisent rendent le passage de l'un vers l'autre encore moins naturel. L'adaptation des MDDs à la programma-

tion par contraintes est également un pas vers l'unification des techniques utilisées en logique propositionnelle et en *model checking*.

Nous introduisons ici une nouvelle relation qui fait le lien de la la programmation par contraintes vers les logiques temporelles.

Bibliographie

[1] Ieee standard for property specification language (psl). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1–182, April 2010.

[2] MagdyS. Abadir, KennethL. Albin, John Havlicek, Narayanan Krishnamurthy, and AndrewK. Martin. Formal verification successes at motorola. *Formal Methods in System Design*, 22(2) :117–123, 2003.

[3] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11) :832–843, November 1983.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[5] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *Revised Lectures from the International Symposium on Compositionality : The Significant Difference*, COMPOS'97, pages 23–60, London, UK, UK, 1998. Springer-Verlag.

[6] Krzysztof R. Apt and Frank J.M. Teusink. Comparing negation in logic programming and in prolog. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[7] KrzysztofR. Apt and Sebastian Brand. Infinite qualitative simulations by means of constraint programming. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, volume 4204 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2006.

[8] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, MosheY. Vardi, and Yael Zbar. The forspec temporal logic : A new temporal property-specification language. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–311. Springer Berlin Heidelberg, 2002.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[10] Philippe Balbiani and Condotta Jean-François. Computational complexity of propositional linear temporal logics based on qualitative spatial or temporal reasoning. In Alessandro Armando, editor, *Frontiers of Combining Systems*, volume 2309 of *Lecture Notes in Computer Science*, pages 162–176. Springer Berlin Heidelberg, 2002.

[11] Behnam Banieqbal and Howard Barringer. Temporal logic with fixed points. In *Temporal Logic in Specification*, pages 62–74, 1987.

[12] Behnam Banieqbal and Howard Barringer. Temporal logic with fixed points. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 62–74. Springer Berlin Heidelberg, 1989.

[13] P. Baptiste, P. Laborie, C. Le Pape, and W. Nuijten. Constraint-based scheduling and planning. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 22. Elsevier, 2006.

[14] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The temporal logic sugar. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer Berlin Heidelberg, 2001.

[15] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue : Past, present and future. *Constraints*, 12(1) :21–62, 2007.

[16] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints*, 18(1) :1–6, 2013.

[17] Hachemi Bennaceur and Mohamed-Salah Affane. Partition-k-ac : An efficient filtering technique combining domain partition and arc consistency. In Toby Walsh, editor, *Principles and Practice of Constraint Programming — CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 560–564. Springer Berlin Heidelberg, 2001.

[18] C Bessiere. *Chapter 3 Constraint Propagation*, volume 2 of *Foundations of Artificial Intelligence*, pages 29–83. Elsevier, 2006.

[19] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1) :1636–1642, 1995.

[20] Alessio Bonfietti, Michele Lombardi, Luca Benini, and Michela Milano. A constraint based approach to cyclic rcpsp. In Jimmy Lee, editor, *Principles and Practice of Constraint Programming – CP 2011*, volume 6876 of *Lecture Notes in Computer Science*, pages 130–144. Springer Berlin Heidelberg, 2011.

[21] Patricia Bouyer, Kim G. Larsen, and Nicolas Markey. Lower-bound-constrained runs in weighted timed automata. *Perform. Eval.*, 73 :91–109, 2014.

[22] Julian Bradfield and Colin Stirling. Chapter 4 - modal logics and mu-calculi : An introduction. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 293 – 330. Elsevier Science, Amsterdam, 2001.

[23] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, C-35(8) :677–691, Aug 1986.

[24] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik und Grundl. Math.*, 6 :66–92, 1960.

[25] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 1020 states and beyond. *Information and Computation*, 98(2) :142 – 170, 1992.

[26] J. R. Büchi. On a decision method in restricted second-order arithmetic. In *International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[27] D.V. Campenhout. Technical report.

[28] Krishnendu Chatterjee, Mickael Randour, and Jean-François Raskin. Strategy synthesis for multi-dimensional quantitative objectives. *Acta Inf.*, 51(3-4) :129–163, 2014.

[29] Hubert Ming Chen. *The Computational Complexity of Quantified Constraint Satisfaction.* PhD thesis, Ithaca, NY, USA, 2004. AAI3140811.

[30] Kenil C. K. Cheng, Jimmy H. M. Lee, and Peter J. Stuckey. Efficient representation of adhoc constraints. In *In Proceedings Of The 18Th International Joint Conference On Artificial Intelligence*, pages 1368–1369, 2003.

[31] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[32] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[33] Broderick Crawford, Carlos Castro, and Eric Monfroy. Solving sudoku with constraint programming. In Yong Shi, Shouyang Wang, Yi Peng, Jianping Li, and Yong Zeng, editors, *Cutting-Edge Research Topics on Multiple Criteria Decision Making*, volume 35 of *Communications in Computer and Information Science*, pages 345–348. Springer Berlin Heidelberg, 2009.

[34] George B. Dantzig. A history of scientific computing. chapter Origins of the Simplex Method, pages 141–151. ACM, New York, NY, USA, 1990.

[35] Rina Dechter. *Constraint processing.* Elsevier Morgan Kaufmann, 2003.

[36] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artif. Intell.*, 49(1-3) :61–95, May 1991.

[37] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3) :250–270, 2001.

[38] Stéphane Demri and Régis Gascon. The effects of bounding syntactic resources on presburger ltl. *J. Log. and Comput.*, 19(6) :1541–1575, December 2009.

[39] Stéphane Demri and Deepak D'Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3) :380 – 415, 2007.

[40] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car sequencing problem in constraint logic programming. In *In European Conference on Artificial Intelligence (ECAI-88*, 1988.

[41] Gregoire Dooms, Yves Deville, and Pierre Dupont. Cp(graph) : Introducing a graph computation domain in constraint programming. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 211–225. Springer Berlin Heidelberg, 2005.

[42] Xavier Dupont, Arnaud Lallouet, Y.C. Law, J.H.M. Lee, and C.F.K. Siu. Programmation par contraintes sur les séquences infinies. In *JFPC 2012*, 2012.

[43] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory.* Perspectives in Mathematical Logic. Springer, 1995.

[44] E. Allen Emerson. Temporal and modal logic. In *Handbook Of Theoretical Computer Science*, pages 995–1072. Elsevier, 1995.

[45] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 169–180, New York, NY, USA, 1982. ACM.

[46] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited : on branching versus linear time (preliminary report). In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 127–140, New York, NY, USA, 1983. ACM.

[47] E. Allen Emerson and Chin-Laung Lei. Modalities for model checking : Branching time logic strikes back. *Sci. Comput. Program.*, 8(3) :275–306, 1987.

[48] Joost Engelfriet and Hendrik Jan Hoogeboom. X-automata on ω-words. *Theor. Comput. Sci.*, 110(1) :1–51, March 1993.

[49] F. Fages. *Programmaton Logique par Contraintes.* Cours de l'Ecole Polytechnique. Ellipses, Paris, 1996.

[50] Bertram Felgenhauer and Frazer Jarvis. Mathematics of sudoku i, 2006.

[51] Fico. Modeling with Xpress. http://www.fico.com/en/request_asset/?asset_id=7952.

[52] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2) :194 – 211, 1979.

[53] Oliver Friedmann and Martin Lange. The modal mu-calculus caught off guard. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 6793 of *Lecture Notes in Computer Science*, pages 149–163. Springer Berlin Heidelberg, 2011.

[54] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 163–173, New York, NY, USA, 1980. ACM.

[55] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[56] Paul Gastin and Denis Oddoux. Fast ltl to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer Berlin Heidelberg, 2001.

[57] Gecode Team. Gecode : Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[58] I.P. Gent and T. Walsh. Csplib : a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http ://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[59] Alfonso E. Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition : PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5–6) :619 – 668, 2009. Advances in Automated Plan Generation.

[60] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman & Hall, 1995.

[61] Carmen Gervet. Interval propagation to reason about sets : definition and implementation of a practical language. *CONSTRAINTS*, pages 191–244, 1997.

[62] Erich Grädel, P. G. Kolaitis, L. Libkin, M. Marx, J. Spencer, Moshe Y. Vardi, Y. Venema, and Scott Weinstein. *Finite Model Theory and Its Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[63] M Gravel, C Gagné, and W L Price. Review and comparison of three methods for the solution of the car sequencing problem. *Journal of the Operational Research Society*, 56(11) :1287–1295, 2005.

[64] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques : A Practical Guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[65] B. A. Trakhtenbrot [T. Hailperin]. Impossibility of an algorithm for the decision problem in finite classes. In *Proceedings of the USSR Academy of Science [American Mathematical Society translations series 2]*, volume 70 [23], pages 569–572 [1–5], (Russian [English]), 1950 [1963].

[66] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag, Berlin, Heidelberg, 2010.

[67] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer Berlin Heidelberg, 1980.

[68] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech : A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1 :460–463, 1997.

[69] Samid Hoda, Willem-Jan van Hoeve, and J.N. Hooker. A systematic approach to mdd-based constraint programming. In David Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, volume 6308 of *Lecture Notes in Computer Science*, pages 266–280. Springer Berlin Heidelberg, 2010.

[70] Samid Hoda, Willem Jan van Hoeve, and John N. Hooker. A systematic approach to mdd-based constraint programming. In *CP*, pages 266–280, 2010.

[71] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[72] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3) :359–411, September 1989.

[73] IBM. ILOG CPLEX.

[74] Frazer Jarvis. Mathematics of sudoku ii. *Mathematical Spectrum*, pages 54–58, 2007.

[75] J. A. W. Kamp. *Tense logic and the theory of order*. PhD thesis, UCLA, 1968.

[76] M. Koubarakis. Temporal csps. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 19. Elsevier, 2006.

[77] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27 :333–354, 1983.

[78] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*, pages 789–840. 1990.

[79] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2) :312–360, March 2000.

[80] Arnaud Lallouet, Yat Chiu Law, Jimmy H.M. Lee, and Charles F.K. Siu. Constraint programming on infinite data streams. In Toby Walsh, editor, *International Joint Conference on Artificial Intelligence*, pages 597–604, 2011.

[81] Leslie Lamport. "sometime" is sometimes "not never" : On the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '80, pages 174–185, New York, NY, USA, 1980. ACM.

[82] Don Lancaster. *TTL cookbook*. Sams, Indianapolis, IN, 1974.

[83] François Laroussinie and Nicolas Markey. Temporal logic with forgettable past. In *In LICS'02*, pages 383–392. IEEE Computer Society Press, 2002.

[84] JasperC.H. Lee and JimmyH.M. Lee. Towards practical infinite stream constraint programming : Applications and implementation. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 8656 of *Lecture Notes in Computer Science*, pages 449–464. Springer International Publishing, 2014.

[85] Daniel Leivant. Higher order logic. In *Handbook of Logic in Artificial Intelligence and Logic Programming (2)*, pages 229–322. 1994.

[86] G. Lenzi. The modal mu-calculus : a survey. *Task Quarterly, Scientific Bulletin of the Academic Computer Centre in Gdansk*, 9(3) :293–316, 2005.

[87] G. Lenzi. Recent results on the modal mu-calculus : a survey. *Rendiconti dell'Istituto di Matematica dell'Università di Trieste*, 42 :235–255, 2010.

[88] R. Lougee-Heimer. The common optimization interface for operations research : Promoting open-source software in the operations research community. *IBM Journal of Research and Development*, 47(1) :57–66, Jan 2003.

[89] N. Markey. *Logiques temporelles pour la vérification : expressivité, complexité, algorithmes.* Thèse de Doctorat d'Université, Université d'Orléans, avril 2003.

[90] K Marriott, P. J. Stuckey, and M Wallace. *Chapter 12 Constraint Logic Programming*, pages 409–452. Foundations of Artificial Intelligence. Elsevier, 2006.

[91] Kenneth Lauchlin McMillan. *Symbolic model checking : an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[92] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In Toby Walsh, editor, *Principles and Practice of Constraint Programming — CP 2001*, volume 2239 of *Lecture Notes in Computer Science*, pages 524–538. Springer Berlin Heidelberg, 2001.

[93] Cédric Pralet Guillaume Infantès Michel Lemaître, Gérard Verfaillie. Approche à base de contraintes pour la synthèse de contrôleur en environnement non déterministe et partiellement observable. In *Actes des 5es Journées Francophones de Planification, Décision et Apprentissage pour la conduite de systèmes (JFPDA 2010)*, 2010.

[94] Cédric Pralet Guillaume Infantès Michel Lemaître, Gérard Verfaillie. Synthèse de contrôleur simplement valide dans le cadre de la programmation par contraintes. In *Actes des 5es Journées Francophones de Planification, Décision et Apprentissage pour la conduite de systèmes (JFPDA 2010)*, 2010.

[95] Roger Mohr and Thomas Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28 :225–233, March 1986.

[96] Roger Mohr and Gérald Masini. Good Old Discrete Relaxation. In Yves Kodratoff, editor, *8th European Conference on Artificial Intelligence (ECAI '88)*, pages 651–656, Munich, Germany, 1988. Pitmann Publishing.

[97] Ugo Montanari and Francesca Rossi, editors. *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings*, volume 976 of *Lecture Notes in Computer Science*. Springer, 1995.

[98] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning : Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[99] BruceD. Parrello, WaldoC. Kabat, and L. Wos. Job-shop scheduling using automated reasoning : A case study of the car-sequencing problem. *Journal of Automated Reasoning*, 2(1) :1–42, 1986.

[100] Fabio Patrizi, Nir Lipoveztky, Giuseppe De Giacomo, and Hector Geffner. Computing infinite plans for ltl goals using a classical planner. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*, IJCAI'11, pages 2003–2008. AAAI Press, 2011.

[101] Marc Pauly and Rohit Parikh. Game logic - an overview. *Studia Logica*, 75(2) :165–182, 2003.

[102] Wuxu Peng and S. Purushothaman Iyer. A new type of pushdown automata on infinite trees. *International Journal of Foundations of Computer Science*, 06(02) :169–186, 1995.

[103] A Peuli and L. Zuck. In and out of temporal logic. In *Logic in Computer Science, 1993. LICS '93., Proceedings of Eighth Annual IEEE Symposium on*, pages 124–135, Jun 1993.

[104] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[105] James F. Power. Thue's 1914 paper : a translation. *CoRR*, abs/1308.5858, 2013.

[106] Cédric Pralet and Gérard Verfaillie. Au delà des QCSP pour résoudre des problèmes de contrôle. In *Huitièmes Journées Francophones de Programmation par Contraintes - JFPC 2012*, Toulouse, France, May 2012.

[107] V. R. Pratt. Semantical considerations on floyd-hoare logic. Technical report, Cambridge, MA, USA, 1976.

[108] A. N. Prior. Modality de dictro and modality de re. *Theoria*, 18 :174–180, 1952.

[109] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[110] Michael O. Rabin. DThomas96languagesecidability of second-order theories and automata on infinite trees. *Bulletin of the American Mathematical Society*, 74 :1025–1029, July 1968.

[111] Alexander Rabinovich. Temporal logics over linear time domains are in pspace. *Inf. Comput.*, 210 :40–67, 2012.

[112] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1) :81–98, Jan 1989.

[113] Grigore Roşu and Saddek Bensalem. Allen linear (interval) temporal logic &#8211 ; translation to ltl and monitor synthesis. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 263–277, Berlin, Heidelberg, 2006. Springer-Verlag.

[114] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[115] Kristin Y. Rozier. Survey : Linear temporal logic symbolic model checking. *Comput. Sci. Rev.*, 5(2) :163–203, May 2011.

[116] J. J.M.M. Rutten. Elements of stream calculus (an extensive exercise in coinduction). Technical report, Amsterdam, The Netherlands, The Netherlands, 2001.

[117] S. Safra. On the complexity of omega -automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, SFCS '88, pages 319–327, Washington, DC, USA, 1988. IEEE Computer Society.

[118] Donald Sannella and Andrzej Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. EATCS Monographs on theoretical computer science. Springer, 2012.

[119] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3 :48–55, 1993.

[120] Paulo Shakarian, Gerardo I. Simari, and V. S. Subrahmanian. Annotated probabilistic temporal logic : Approximate fixpoint implementation. *ACM Trans. Comput. Logic*, 13(2) :13 :1–13 :33, April 2012.

[121] E. Shapiro. Systolic programming : A paradigm of parallel processing. In E. Shapiro, editor, *Concurrent Prolog : Collected Papers (Volume I)*, pages 207–242. MIT Press, London, 1987.

[122] H. Simonis. Sudoku as a constraint satisfaction problem, 2005.

[123] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 159–168, New York, NY, USA, 1982. ACM.

[124] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3) :733–749, July 1985.

[125] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2–3) :217 – 237, 1987.

[126] Fai Keung Siu. *Constraint Programming on Infinite Data Streams*. PhD thesis, 2012. AAI3570338.

[127] Barbara Smith. CSPLib problem 001 : Car sequencing. http://www.csplib.org/Problems/prob001.

[128] Gert Smolka and Ralf Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3) :229 – 258, 1994.

[129] Christine Solnon, Van Dat Cung, Alain Nguyen, and Christian Artigues. The car sequencing problem : overview of state-of-the-art methods and industrial case-study of the roadef'2005 challenge problem. *European Journal Of Operational Research (EJOR)*, 2007.

[130] L. J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis.

[131] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. Doctoral dissertation, Saarland University, 2009.

[132] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

[133] Wolfgang Thomas. Handbook of theoretical computer science (vol. b). chapter Automata on infinite objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.

[134] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[135] Edward P. K. Tsang. *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press, 1993.

[136] P. van Beek. Backtracking search algorithm. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4. Elsevier, 2006.

[137] M. Y. Vardi. A temporal fixpoint calculus. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 250–259, New York, NY, USA, 1988. ACM.

[138] Moshe Y. Vardi. Automata-theoretic techniques for temporal reasoning. In *In Handbook of Modal Logic*, pages 971–989. Elsevier, 2006.

[139] Moshe Y. Vardi. Pillars of computer science. chapter From Monadic Logic to PSL, pages 656–681. Springer-Verlag, Berlin, Heidelberg, 2008.

[140] Moshe Y. Vardi and Larry J. Stockmeyer. Improved upper and lower bounds for modal logics of programs : Preliminary report. In Robert Sedgewick, editor, *STOC*, pages 240–251. ACM, 1985.

[141] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2) :183 – 221, 1986.

[142] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115 :1–37, 1994.

[143] MosheY. Vardi. An automata-theoretic approach to linear temporal logic. In Faron Moller and Graham Birtwistle, editors, *Logics for Concurrency*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Berlin Heidelberg, 1996.

[144] MosheY. Vardi. Branching vs. linear time : Final showdown. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2001.

[145] MosheY. Vardi. The büchi complementation saga. In Wolfgang Thomas and Pascal Weil, editors, *STACS 2007*, volume 4393 of *Lecture Notes in Computer Science*, pages 12–22. Springer Berlin Heidelberg, 2007.

[146] M.Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 327–338, Oct 1985.

[147] Gérard Verfaillie and Cédric Pralet. Constraint programming for controller synthesis. In *CP*, pages 100–114, 2011.

[148] Marc Vilain, Henry Kautz, and Peter van Beek. Readings in qualitative reasoning about physical systems. chapter Constraint Propagation Algorithms for Temporal Reasoning : A Revised Report, pages 373–381. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[149] Klaus Weihrauch. *Computable Analysis*. Texts in Theoretical Computer Science. Springer, 2000.

[150] P. Wolfe. The composite simplex algorithm. *SIAM Review*, 7(1) :42–54, 1965.

[151] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2) :72–99, 1983.

[152] Pierre Wolper, M.Y. Vardi, and APrasad Sistla. Reasoning about infinite computation paths. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 185–194, Nov 1983.

[153] Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In Günther Görz and Steffen Hölldobler, editors, *KI-96 : Advances in Artificial Intelligence*, volume 1137 of *Lecture Notes in Computer Science*, pages 377–386. Springer Berlin Heidelberg, 1996.

[154] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 17–36, London, UK, UK, 2002. Springer-Verlag.